Memory-Efficient HQC Implementations on Cortex-M3 and Cortex-M7

Minjoo Sim[1][0000-0001-5242-214X], Siwoo Eum[1][0000-0002-9583-5427], Minwoo Lee[1][0000-0002-2356-3055], Seungwon Lee[1][0009-0009-9583-1387], Gyeongju Song[1][0000-0002-4337-1843], Lee Wai Kong[2][0000-0003-4659-8979], Anupam Chattopadhyay[3][0000-0002-8818-6983], and Hwajeong Seo[1][0000-0003-0069-9061] ⋆

[1] Hansung University, Seoul, Republic of Korea
{minjoos9797,shuraatum,minunejip}@gmail.com
{dkajdfhd1,thdrudwn98,hwajeong84}@gmail.com
[2] Universiti Tunku Abdul Rahman, Kampar, Malaysia
wklee@utar.edu.my
[3] Nanyang Technological University, Singapore
anupam@ntu.edu.sg

Abstract. In the Hamming Quasi-Cyclic (HQC) key encapsulation mechanism (KEM), eliminating the polynomial product buffer still leaves serialized inputs, operands, and candidate ciphertexts that may coexist during decapsulation. We organize storage around the last use of each value. Comparing each candidate component before producing the next allows both to share one workspace, while deterministic regeneration shortens the lifetimes of ephemeral supports. Our lifetime analysis relates the measured footprint to the values that coexist during reencryption in our compact design. We read dense operands directly from serialized inputs and reuse output storage as workspace before final serialization during key generation and encapsulation. These choices reduce overlapping storage across the complete KEM and complement established support-form and Zero-Buffer multiplication. We evaluate this organization for all three HQC parameter sets on ARM Cortex-M3 and Cortex-M7 under the August 2025 specification. In a cumulative Cortex-M3 HQC-256 comparison, combined representation and lifetime changes reduced the stack by 38.9 % relative to a Zero-Buffer configuration that already uses component-wise ciphertext comparison. Using a separate configuration, we measured decapsulation footprints of 36,858 bytes on Cortex-M3 and 36,850 bytes on Cortex-M7, including peak operation stack and serialized input/output storage. The common object layout accounts for most of these footprints. In the evaluated configurations, the relative increase in decapsulation cycles from the direct to the compact implementation is larger on Cortex-M7 than on Cortex-M3. Retaining supports reduces cycles when memory permits.

⋆ Corresponding author. hwajeong84@gmail.com

The kernels retain support-dependent control flow and memory accesses; we did not establish complete-KEM constant-time execution or physical side-channel resistance.

Keywords: HQC · ARM Cortex-M · Embedded implementation · Post-quantum cryptography · Code-based cryptography

1 Introduction

Hamming Quasi-Cyclic (HQC) is a code-based key encapsulation mechanism (KEM) selected by the National Institute of Standards and Technology (NIST) for standardization alongside the Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM) [39,17,38]. Its polynomial operands and Reed–Muller and Reed–Solomon decoder state grow with the parameter dimensions in Table 1. An embedded application must fit these objects within the SRAM budget available to the KEM, leaving space for the remaining firmware. This requires controlling which objects coexist across key generation, encapsulation, and decapsulation, as well as optimizing the underlying arithmetic.

During decapsulation, serialized inputs, parsed and candidate ciphertexts, polynomial workspace, and decoder state may coexist. Reducing the multiplier's workspace leaves the storage occupied by these other values to be addressed. We organize the complete KEM around their lifetimes: we consume serialized inputs directly, compare candidate ciphertext components before reusing their storage, and regenerate temporary values when retaining them would extend their lifetimes across KEM phases. This organization makes the storage shared between decryption and reencryption central to the design.

Most optimized Cortex-M HQC implementations target the Cortex-M4 [2,29,25,10,34]. MUTUALISM combines multiplication using lists of nonzero positions (supports) with complete-KEM memory sharing and also reports Cortex-M33 results [34]. We chose Cortex-M3 and Cortex-M7 to evaluate the same memory organization on cores with different digital signal processing (DSP) support, pipelines, and floating-point scratch registers (Section 2.3).

Prior Cortex-M3 code-based implementations include ROLLO-I and a decrypt-only Classic McEliece port [30,50]. Šrubař benchmarks every PQClean HQC-128 KEM operation on Arduino Due; HQC-192 exceeds that board's memory, and HQC-256 is not evaluated there [45].

We compare Direct-Asm, which uses support-form multiplication with a double-length product buffer, with Compact-Asm, which combines lifetime-based data flow, Zero-Buffer multiplication, and deterministic support regeneration [2,14,34]. Direct serialized access and lifetime-based sharing also appear in MUTUALISM [34]; Section 2.5 compares the two data flows.

We evaluate every KEM operation for all three parameter sets on Cortex-M3 and Cortex-M7. We measured Compact-Asm operation footprints with `-Os` for HQC-128/192 and `-O3` for HQC-256 (Tables 9 and 10). Footprint includes complete-operation peak stack use and serialized input/output storage; Section 4 accounts for the remaining resident data.

Table 1: HQC parameters, NIST security levels, and serialized sizes in bytes for the August 2025 specification.

Scheme	Level	n	(n_1, n_2, δ)	$(\omega, \omega_r, \omega_e)$	pk	sk	ct
HQC-128	1	17,669	(46, 384, 15)	(66, 75, 75)	2241	2321	4433
HQC-192	3	35,851	(56, 640, 16)	(100, 114, 114)	4514	4602	8978
HQC-256	5	57,637	(90, 640, 29)	(131, 149, 149)	7237	7333	14,421

1.1 Contributions

Complete-KEM Memory Organization and Analysis. We organize decryption and reencryption around the last use of each value, combining serialized-input access, component-wise ciphertext comparison, and support regeneration with Zero-Buffer multiplication. The lifetime analysis identifies the reuse conditions and the objects that coexist (Section 3.5). For HQC-256 decapsulation, the serialized API objects, two ring vectors, and retained support occupy 36,500 bytes in the present allocation, accounting for most of the measured footprint. In a cumulative Cortex-M3 HQC-256 comparison with `-Os` and SRAM-resident decoder tables, combined representation and lifetime changes reduced the decapsulation stack by 9584 bytes relative to a configuration already using Zero-Buffer multiplication and component-wise ciphertext comparison (Table 12).

Evaluation on Two Cortex-M Cores. We evaluate all three parameter sets and every KEM operation on Cortex-M3 and Cortex-M7 using the same compact memory organization. The measured HQC-256 decapsulation footprints are nearly identical on the two cores. For the evaluated configurations, the relative increase in decapsulation cycles from Direct-Asm to Compact-Asm is greater on Cortex-M7 across all three parameter sets (Section 4).

Storage–Recomputation Trade-off. We quantify the cost of shortening support lifetimes by comparing retention and regeneration within the compact design, with the remaining transformations fixed (Table 18). In the tested HQC-256 decapsulation configurations, regeneration saves 588 and 600 bytes of stack on Cortex-M3 and Cortex-M7 with cycle increases of 2.27 % and 8.35 %, respectively.

Section 2.4 defines the timing and memory-placement assumptions and security scope of the evaluation.

Organization. Section 2 introduces HQC, multiplication methods, and the targets. Section 3 presents the implementations and core-specific optimizations. Section 4 evaluates the two primary targets and compares the resulting design with Cortex-M4 work. Section 5 concludes.

2 Preliminaries

2.1 HQC

We implemented and evaluated HQC according to the August 22, 2025 specification [17]. The October 12, 2026 revision increases selected fixed weights, makes the seed-only decapsulation key the default, and changes the irreducible polynomial for $\mathbb{F}_{2^8}$ [23]. These changes affect support sizes and processing, key expansion, and decoder field tables. The direct access to an embedded public key described below requires an expanded key; a seed-only key requires regeneration or cached expanded state.

HQC combines the Fujisaki–Okamoto transform with a quasi-cyclic public-key encryption scheme [22]. Its ring arithmetic uses $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$, where addition is bitwise XOR and multiplication is cyclic binary convolution. HQC combines a Reed–Solomon (RS) code over $\mathbb{F}_{2^8}$ with a duplicated Reed–Muller (RM) code RM$(1, 7)$.

The outer RS code maps the $k/8$ message bytes to n_1 symbols over $\mathbb{F}_{2^8}$, where k is the message length in bits. Each symbol is encoded with the $[128, 8, 64]$ code RM$(1, 7)$; the resulting codeword is duplicated three times for HQC-128 or five times for HQC-192 and HQC-256, giving $n_2 \in \{384, 640\}$. Encode applies RS encoding followed by duplicated RM encoding, and Decode reverses these steps. The specified ring length n is prime and exceeds $n_1 n_2$. Ring products cover all n coordinates, whereas $\mathsf{Truncate}$ retains only the $n_1 n_2$ coordinates used by the code.

Key generation expands the secret seed into fixed-weight vectors x and y of weight ω, and the public seed into the dense vector h. The public polynomial is $s = x + hy$. Encryption derives fixed-weight vectors r_1, r_2, and e from the message randomness. It computes

$$\begin{aligned} u &= r_1 + hr_2, \\ v &= \mathsf{Encode}(m) + \mathsf{Truncate}(sr_2 + e). \end{aligned}$$

Decryption first computes $w = v + \mathsf{Truncate}(uy)$ and decodes w. Decapsulation then repeats encryption with the recovered message and compares the resulting ciphertext with its input.

Let ek_{KEM} denote the public encapsulation key, c the ciphertext, and σ the secret used for implicit rejection. We denote the specification hash functions by H, G, and J. The value $\mathsf{G}(\mathsf{H}(ek_{\mathrm{KEM}}), m, \mathsf{salt})$ yields the shared key K and encryption seed θ, while $\mathsf{J}(\mathsf{H}(ek_{\mathrm{KEM}}), \sigma, c)$ yields the implicit rejection key $\bar{K}$. Decapsulation uses the recovered $m^\star$ to obtain $(K^\star, \theta^\star)$, compares $c^\star$ with c, and selects either $K^\star$ or $\bar{K}$.

HQC separates its public-key encryption (PKE) layer from the KEM layer. The PKE keys are $ek_{\mathrm{PKE}} = (\mathsf{seed}_{ek}, s)$ and $dk_{\mathrm{PKE}} = \mathsf{seed}_{dk}$. The serialized KEM secret key contains ek_{KEM}, dk_{PKE}, the implicit rejection secret σ, and the KEM seed. Our implementation does not serialize x or y. During decapsulation, it reconstructs the support of y from dk_{PKE} and uses the embedded ek_{KEM} for reencryption.

We write $\mathsf{SampleFW}_1$ for the specification-defined rejection sampler that regenerates the long-term vectors x and y from their seed. We retained the reference out-of-range and duplicate checks and the extendable-output function (XOF) consumption order. The fixed-work $\mathsf{SampleFW}_2$ sampler produces ephemeral r_1, r_2, and e [44]. Each sampler returns a support of the corresponding public weight in Table 1. We follow the specified sampler call order and preserve the seed-to-support mapping of the evaluated reference implementation. For ephemeral sampling, this requires its multiplicative range conversion, which differs from the modular conversion in the specification's sampling pseudocode.

Algorithm 1 gives independent sketches of the dominant ring products: one in key generation, two in encapsulation, and three in decapsulation, including reencryption. It includes reencryption key derivation but omits the remaining key derivation and serialization steps. The XOF state ctx supplies successive sampler calls in order. Each product combines a dense operand (h, s, or u) with the support of y or r_2. We focus the lifetime analysis on decapsulation, which retains the input and decoded message while constructing a comparison ciphertext (Section 3.5).

$\mathsf{Expand}(\mathsf{seed}_{ek})$ deterministically expands the public seed into h. $\mathsf{SFProd}(a, S)$ is the support-form product of dense operand a and support S. $\mathsf{Dense}(S)$ denotes the corresponding mathematical fixed-weight vector, which need not be allocated in dense form. When a fixed-weight operand is stored in support form, the implementation adds the operand by directly toggling the coordinates in S.

Table 1 gives the public dimensions that determine accumulator bounds and persistent-buffer sizes. Here n_1 is the number of outer RS symbols, n_2 the duplicated RM length, and δ the RS correction capacity. The weights $(\omega, \omega_r, \omega_e)$ apply to (x, y), (r_1, r_2), and e. The public-key (pk), secret-key (sk), and ciphertext (ct) columns report the serialized KEM sizes.

2.2 Multiplication choices

A fixed-weight operand can be represented densely or by its support.

- **Dense multiplication.** The reference multiplies two dense vectors. Karatsuba and comb methods use regular memory accesses but require scratch space proportional to the polynomial length.
- **FAFFT.** Chen et al. apply the Frobenius additive fast Fourier transform (FAFFT) [33] to both operands. Their Cortex-M4 assembly uses packed arithmetic, DSP instructions, and floating-point scratch registers. The regular transform requires tables and workspace.
- **CRT.** Chinese remainder theorem (CRT) decomposition uses pointwise products in a tower field; recombination adds data movement and temporary state [10].
- **Support-form multiplication.** Storing a fixed-weight operand as its nonzero positions turns multiplication into the XOR of shifted dense copies [14,2]. The public weight fixes the iteration count. Word loads, shifts, and XOR replace arithmetic products and avoid recursive or transform scratch.

Algorithm 1: Independent sketches of the dominant ring-product calls in each HQC KEM operation.

Key generation
  Given Public and secret seeds seed_{ek} and seed_{dk}
  Result Public polynomial s
  $\mathsf{ctx}_{dk} \leftarrow \mathsf{XOF}(\mathsf{seed}_{dk})$
  $S_y \leftarrow \mathsf{SampleFW}_1(\mathsf{ctx}_{dk}, \omega)$
  $S_x \leftarrow \mathsf{SampleFW}_1(\mathsf{ctx}_{dk}, \omega)$
  $h \leftarrow \mathsf{Expand}(\mathsf{seed}_{ek})$
  $s \leftarrow \mathsf{Dense}(S_x) \oplus \mathsf{SFProd}(h, S_y)$

Encapsulation
  Given Public polynomials h, s, message m, and encryption seed θ
  Result Ciphertext components u, v
  $\mathsf{ctx}_{\theta} \leftarrow \mathsf{XOF}(\theta)$
  $S_{r_2} \leftarrow \mathsf{SampleFW}_2(\mathsf{ctx}_{\theta}, \omega_r)$
  $S_e \leftarrow \mathsf{SampleFW}_2(\mathsf{ctx}_{\theta}, \omega_e)$
  $S_{r_1} \leftarrow \mathsf{SampleFW}_2(\mathsf{ctx}_{\theta}, \omega_r)$
  $u \leftarrow \mathsf{Dense}(S_{r_1}) \oplus \mathsf{SFProd}(h, S_{r_2})$
  $v \leftarrow \mathsf{Encode}(m) \oplus \mathsf{Truncate}(\mathsf{SFProd}(s, S_{r_2}) \oplus \mathsf{Dense}(S_e))$

Decapsulation
  Given Secret key and received ciphertext c, including u, v and salt
  Result Shared key
  $\mathsf{ctx}_{dk} \leftarrow \mathsf{XOF}(\mathsf{seed}_{dk})$
  $S_y \leftarrow \mathsf{SampleFW}_1(\mathsf{ctx}_{dk}, \omega)$
  $w \leftarrow v \oplus \mathsf{Truncate}(\mathsf{SFProd}(u, S_y))$
  $m^{\star} \leftarrow \mathsf{Decode}(w)$
  $(K^{\star}, \theta^{\star}) \leftarrow \mathsf{G}(\mathsf{H}(ek_{\mathrm{KEM}}), m^{\star}, \mathsf{salt})$
  derive $S^{\star}_{r_2}$ and $S^{\star}_e$ with the first two $\mathsf{SampleFW}_2$ calls
  derive $S^{\star}_{r_1}$ with the next call in the same XOF stream
  recompute $u^{\star}$ and $v^{\star}$ with the two encapsulation products
  compare the ciphertexts and select the shared key

Transform methods benefit from Cortex-M4 instructions when SRAM permits. We use support-form multiplication for every KEM ring product, with processor-specific schedules in Direct-Asm and a shared four-word schedule in Compact-Asm (Section 3). Direct-Asm reduces a double-length buffer after the support loop; Compact-Asm accumulates into its destination in two passes separated by an in-place reduction. Only Compact-Asm removes full-product scratch. Both retain support-derived accesses, and the compact partition bounds depend on the support.

Fixed-weight sampling still uses integer products for range conversion, and the decoder retains field multiplication. Section 3 discusses these costs separately from the ring accumulator.

Table 2: Target processors.

Core	Board	Long mul.	Core D-cache	Evaluated memory	Role
Cortex-M3	F207ZG	Variable	None	128 KiB SRAM	Primary
Cortex-M7	F767ZI	Fixed	16 KiB	128 KiB DTCM	Primary

2.3 Target processors

Table 2 lists the architectural features relevant to the two implementations. We used the F207ZG and F767ZI as the primary targets [47,46]. Published Cortex-M4 results provide comparison context. The Cortex-M7 writable working state resides in data tightly coupled memory (DTCM).

The Cortex-M3 lacks the floating-point registers used as scratch by the Cortex-M4 transform and has variable-latency long multiplication. The Cortex-M7 provides packed arithmetic and fixed-latency long multiplication, and its pipeline can issue suitable independent load/arithmetic pairs [3,4,5]. We used a different direct-product schedule on each core. The compact Cortex-M7 implementation uses the same Zero-Buffer schedule as Cortex-M3 (Section 3.3).

2.4 Threat and memory-placement model

We distinguish remote and cycle-count timing, cache timing, and physical power or electromagnetic leakage. Timing effects include operand-dependent instruction latency and support-derived control flow; cache-timing effects include secret-dependent indexing and replacement. Wafo-Tapa et al. exploited error-weight-dependent BCH decoding time in an earlier HQC version [51]. We consider decoding separately from multiplication; Section 3.6 describes the RS/RM decoder used here. We did not evaluate physical leakage of the kind studied in earlier HQC attacks [41,49]. Recent physical attacks exploit sparse-vector load/store leakage on Cortex-M4 [6], repeated legitimate decryption in non-SIMD HQC [35], and single-trace leakage from protected Cortex-M4 implementations [16]. Keeping data out of the cache does not remove the intermediate values or load/store activity targeted by these attacks.

Huang et al. exploited cache accesses during the conversion of sparse supports to dense vectors [24]. Those addresses depend on support coordinates even when the public weight fixes the loop count. We placed writable working state and caller-owned input/output buffers in cacheless SRAM on Cortex-M3 and in DTCM on Cortex-M7. Direct serialized access reads ciphertext and secret-key data from these caller buffers. These placements avoid data-cache indexing and replacement for the writable objects in our measurements; they do not hide support-derived control flow or address traces. The decoder also uses secret-dependent field-table indices. In the direct–compact cycle comparison, we placed these tables in flash for Compact-Asm. Section 4.1 reports the placements used in each measurement configuration. The F207 includes a flash data cache, and F767 flash accesses can use the core data cache [47,46]; placing the tables in

read-only memory preserves their secret-dependent indexing. The direct multiplier has fixed loop bounds. The compact Zero-Buffer multiplier performs a fixed total number of limb updates across its partitions, but each partition's bound depends on the support. Neither implementation is address-oblivious.

For a fixed key, the candidate count of $\mathsf{SampleFW}_1$ is independent of the ciphertext, although it can vary across keys. This differs from the ciphertext-dependent rejection sampling studied by Guo et al. [20]. During reencryption, the ephemeral supports depend on $\theta^\star$, which we derive from the decoded message $m^\star$ and received salt. The support-dependent multiplication paths can therefore vary with the ciphertext even though $\mathsf{SampleFW}_2$ has fixed loop counts. Lai et al. [31] study passive leakage from sparse-vector processing that does not require ciphertext-dependent sampling counts. Hesse et al. [21] demonstrate power leakage in fixed-weight sampling and support conversion, including a masked software implementation.

Compiler-induced leakage in optimized HQC shows that source-level mask-based selection alone does not establish constant-time binary behavior [15]. We evaluated Cortex-M3 cycle variation with valid and invalid ciphertexts (Section 4.6). This comparison does not isolate the dependencies on decoded messages and reencryption supports, so it cannot establish their absence from the timing trace. We make no complete-KEM constant-time or physical side-channel resistance claim.

2.5 Related work

Code-based schemes on Cortex-M include Classic McEliece and BIKE [11,12]. QC-MDPC implementations such as QcBits also use support-driven rotations and accumulations [13].

Published Cortex-M4 results span several boards and specification generations. They include HQC-128 benchmarks [28,48] and FAFFT/CRT implementations [10]. Aissaoui et al. provide a support-form HQC-128 implementation; the subsequent thesis also measures HQC-192 [2,1]. Lee et al. optimize FAFFT linear layers and register scheduling for HQC and BIKE [32]; Kim et al. and Jang et al. optimize polynomial multiplication [29,25]. Subsequent work reduces FAFFT register transfers, optimizes support expansion, and evaluates transform and hash caching for public-key reuse [26]. Strided FAFFT uses smaller fields and reorganized transforms on Cortex-M4 and wider processors [9]. MUTUALISM presents a low-memory design for Cortex-M4 and Cortex-M33 [34].

Table 3 compares target platforms, representations, and parameter-set coverage. Chen et al. provide the transform-based comparison; MUTUALISM is the closest low-memory Cortex-M4 comparison. Our compact design and MUTUALISM both use support-form products, Zero-Buffer accumulation, direct access to serialized polynomials, and lifetime-based buffer sharing. MUTUALISM also uses incremental SHAKE absorption to remove concatenated hash-input buffers [34]. Component-wise reencryption comparison also appears in the Kyber implementation of Botros et al. [7,8]. We apply component-wise comparison and deterministic support regeneration to the complete HQC data flow

Table 3: Scope of Cortex-M HQC implementations.

Work	Target	Parameter sets	Representation	Coverage
pqm4 [28]	F407 (M4)	HQC-128	Dense	Complete KEM
Tasopoulos et al. [48]	F439ZI (M4)	HQC-128	Dense	TLS and KEM
Aissaoui et al. [2]	F407 (M4)	HQC-128	Support	Complete KEM
Aissaoui thesis [1]	F407 (M4)	HQC-128/192	Support	Complete KEM
Šrubař [45]	Due (M3)	HQC-128	Dense	Complete KEM
Chen et al. [10]	L4R5ZI (M4)	All	FAFFT and CRT	Complete KEM
MUTUALISM [34]	L4R5ZI/MG24	All	Support	Complete KEM
This work	F207ZG/F767ZI	All	Support	Complete KEM

on Cortex-M3 and Cortex-M7. The analysis in Section 3.5 identifies the reuse conditions and relates the coexisting objects to the measured footprint. It also explains how we retain a detected ciphertext mismatch and preserve hash inputs and the final shared-secret selection rules when reusing storage. We compared support retention and regeneration with the remaining compact transformations held fixed (Table 18). Table 9 places the published MUTUALISM results alongside our measurements, subject to the cross-study limitations in Section 4.

We focus on complete software implementations on unmodified Cortex-M microcontrollers.

3 Memory-Efficient HQC Implementation

We organize the implementation around the objects that coexist during a complete KEM operation. We combine Zero-Buffer multiplication with direct access to serialized operands and reuse intermediate storage after its last use, avoiding redundant imports and shortening the lifetimes of candidate ciphertexts and ephemeral supports. We first describe the operand representations and multiplication methods, then the processor-specific schedules and complete-KEM buffer lifetimes.

3.1 Complete HQC call flow and representation

The reference implementation materializes fixed-weight operands and multiplies dense polynomials with recursive Karatsuba. Its full-product and recursion arrays alone occupy 39,888, 80,784, and 129,744 bytes for HQC-128, HQC-192, and HQC-256, respectively. These arrays do not account for decoder state, serialized application programming interface (API) objects, or the caller stack. We therefore chose representations that reduce operand storage throughout the complete call flow.

In Compact-Asm, we store x, y, r_1, r_2, and e as lists of 16-bit positions on both cores and for all parameter sets. This representation is lossless because every HQC ring dimension is below 2^{16}. We pass each support directly to the Zero-Buffer multiplier or expand it into a destination by toggling one bit at

each listed position. We preserve the XOF consumption order of the corresponding dense implementation, without allocating a dense conversion buffer. Public keys, secret keys, and ciphertexts remain in the serialized formats defined by the specification.

We also avoid retaining polynomial copies when their serialized values can be consumed directly. Section 3.5 describes how we combine these reads with caller-owned workspaces and immediate comparison of candidate ciphertext components. Figure 3 shows the resulting decapsulation flow.

3.2 Support-form multiplication

Let a be a canonical dense polynomial, $\deg a < n$, and let $S = (p_0, \ldots, p_{\omega-1})$ contain the nonzero positions of a fixed-weight polynomial, with $0 \le p_k < n$. Their product is

$$o = \sum_{k=0}^{\omega-1} aX^{p_k} \bmod (X^n - 1). \tag{1}$$

Equation 1 expresses the product as one shifted copy of the dense operand for each support position. The public weight fixes the number of processed positions; each position and its derived word and bit offsets remain secret. We establish the accumulator invariants below for these operands, whose unreduced product has degree at most $2n - 2$. We discuss noncanonical serialized operands separately at the reduction boundary.

Direct support-form multiplication Direct-Asm stores the dense operand in $M_{32} = 2\lceil n/64 \rceil$ zero-padded 32-bit limbs and accumulates every linear shift in a $2M_{32}$-limb buffer. Figure 1 shows this layout. After the support loop, the implementation folds bits at or above degree n. This implementation provides the comparison in Tables 14 and 16. The compact implementation accumulates directly in its destination instead of this double-length buffer.

Algorithm 2 separates fixed loop bounds from secret-derived addresses. Lines 4 and 8 use the public bounds ω and M_{32}. Lines 5–6 derive word and bit offsets from a support position, and line 9 updates the corresponding accumulator address. The loop maintains the following invariant: after processing p_k, the linear accumulator equals the XOR of all unreduced shifts processed so far. The carry from the two-shift expression is zero when $\beta = 0$, and zero padding provides space for every linear shift. The reduction modulo $X^n - 1$ folds the upper part, after which the final mask clears unused bits in the partial destination limb.

Zero-Buffer compact multiplication We use two-pass Zero-Buffer support multiplication in Compact-Asm [34], reading dense operands directly from their serialized form and writing results to caller-owned destinations (Section 3.5). Both cores use a four-word assembly schedule. Let $L_{32} = \lceil n/32 \rceil$ and $q = \lfloor p_k/32 \rfloor$. In the first pass, each support position accumulates q limbs from the upper part of the dense operand into the start of the output, including the incoming carry

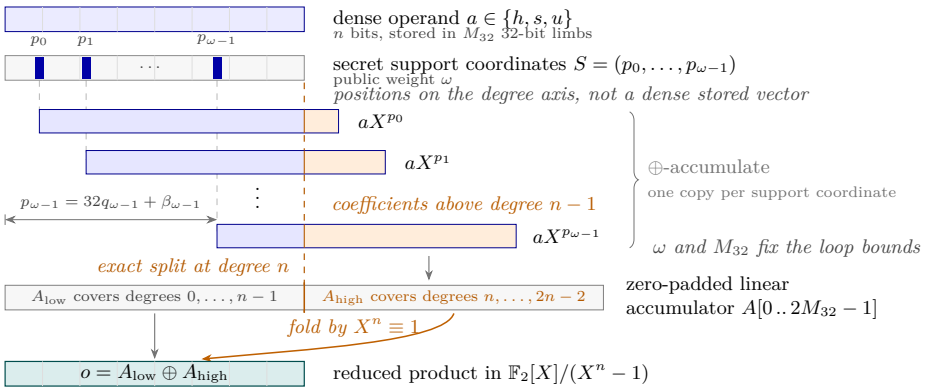

Fig. 1: Linear accumulation and reduction in Direct-Asm.

from the preceding limb and the outgoing carry at limb q. After all support positions have been processed, a whole-output shift with public bounds performs the intermediate reduction. The second pass processes all support positions again, accumulating the remaining $L_{32} - q$ limbs into the output starting at limb q. For each support position, the two secret-derived loop lengths sum to the public constant L_{32}. A final fold and mask complete the reduction in o, without the double-length buffer A.

We write the direct accumulator polynomial as $A(X) = A_{\text{lo}}(X) + X^{32L_{32}} A_{\text{hi}}(X)$, where $\deg A_{\text{lo}} < 32L_{32}$. After the first pass, o contains A_{hi}. The incoming carry preserves contributions that cross the split. For a support position with $q = 0$, the upper-part contribution consists only of this carry. The whole-output shift multiplies this state by $X^{32L_{32}-n}$, and the second pass adds A_{lo}. Thus, before the final fold,

$$o(X) = A_{\text{lo}}(X) + X^{32L_{32}-n} A_{\text{hi}}(X) \equiv A(X) \pmod{X^n - 1}.$$

Both passes use the same shift-and-carry operation as Direct-Asm. For each ℓ-limb segment, we process $\lfloor \ell/4 \rfloor$ four-word blocks followed by $\ell \bmod 4$ scalar updates, carrying the same intermediate word across the two loops. The segment lengths, and hence the block and scalar-tail counts, depend on the support. The fixed limb count therefore does not imply a fixed instruction count. We form the final partial input limb from bytes within its serialized field, avoiding a dense import buffer. The split point, per-pass loop bounds, effective addresses, bit shifts, and partial-limb branch also depend on the support. Section 2.4 discusses these dependencies and the resulting security scope.

Fixed-ω iteration work In Compact-Asm, each product performs ωL_{32} or $\omega_r L_{32}$ dense-limb updates; decapsulation computes uy and two reencryption products (Table 4 and Algorithm 1).

Algorithm 2: Direct support-form multiplication.

Input Canonical operand a in zero-padded limbs $a[0 \dots M_{32} - 1]$, support $S = (p_0, \dots, p_{\omega-1})$, public weight ω, and public M_{32}
Output Product o
$A[0 \dots 2M_{32} - 1] \leftarrow 0$
for $k \leftarrow 0$ **to** $\omega - 1$ **do**
  $q \leftarrow \lfloor S[k]/32 \rfloor$
  $\beta \leftarrow S[k] \bmod 32$
  $c \leftarrow 0$
  for $j \leftarrow 0$ **to** $M_{32} - 1$ **do**
    $A[q + j] \leftarrow A[q + j] \oplus ((a[j] \ll \beta) \bmod 2^{32}) \oplus c$
    $c \leftarrow (a[j] \gg 1) \gg (31 - \beta)$
  $A[q + M_{32}] \leftarrow A[q + M_{32}] \oplus c$
$o \leftarrow \mathsf{Reduce}_{X^n-1}(A)$
return o

Table 4: Zero-Buffer fixed-ω main-loop work in 32-bit limb updates.

Parameter	L_{32}	Key generation	Encapsulation	Decapsulation
HQC-128	553	36,498	82,950	119,448
HQC-192	1,121	112,100	255,588	367,688
HQC-256	1,802	236,062	536,996	773,058

The counts in Table 4 apply to both cores and exclude per-support carry handling and the public whole-output shift. Direct-Asm instead uses the padded M_{32} count. Update costs depend on the kernel and processor; complete-operation time also includes initialization, carry handling, reduction, support generation, decoding, hashing, and serialization.

Reduction boundary Compact-Asm treats dense operands and outputs as arrays of 32-bit limbs. The whole-output shift folds the upper part in place. Let $r = n \bmod 32$, which is nonzero for all three parameter sets. After the second pass, it folds the remaining coefficients of degrees $n, \dots, 32L_{32} - 1$ into the first limb and then masks the last limb:

$$o[0] \leftarrow o[0] \oplus (o[L_{32} - 1] \gg r),$$
$$o[L_{32} - 1] \leftarrow o[L_{32} - 1] \,\&\, (2^r - 1).$$

This fold uses $X^n = 1$ to map the remaining high coefficients to the low limb. We form the last partial input limb from bytes within the serialized field and set the bytes beyond its boundary to zero. We retain unused bits inside the last serialized byte until product reduction. These bits can produce terms of degree at least $2n$, for which the compact and reference reductions differ. For HQC-256, $a = X^{n+1}$ and a support containing $n - 1$ produce an X^{2n} term. The compact fold maps this term to the constant coefficient, whereas the reference

Table 5: Architecture-specific multiplication and decoder schedules.

Core	Direct-Asm	Compact-Asm	Decoder operations
Cortex-M3	Four-word block transfer	Four-word block transfer	Scalar halfwords
Cortex-M7	Two-word paired schedule	Four-word block-transfer kernel	Packed DSP

and direct reductions fold it once and discard it through the final mask. We therefore restrict the arithmetic-equivalence argument to canonical operands. We compared shared-secret outputs in Section 4.1; those checks do not establish equality of intermediate products for noncanonical encodings.

3.3 Architecture-specific support implementations

The C support-form implementation, Opt-C, stores 32-bit support positions, accumulates 64-bit shifts in a double-length buffer, and then reduces the product. We wrote assembly for the support accumulator, vector, and decoder loops; Table 5 summarizes the processor-specific schedules. The vector XOR, RS syndrome, and RM butterfly loop bounds depend only on public dimensions. Small initialization loops remain in C, and Keccak uses existing assembly routines.

Cortex-M3 Cortex-M3 provides neither DSP instructions nor floating-point registers for scratch storage, and its long multiplier has variable latency. Our accumulator uses block transfers. In the sampler range conversions, we replace long multiplication with fixed 16-bit partial products.

Support-form accumulator In Direct-Asm on Cortex-M3, we process four words in each main-loop iteration. We load four accumulator words with `LDMIA`, read four dense words, keep the cross-word carry in a register, and store four results with `STMIA`. Figure 2(a) shows this schedule. The input and accumulator pointers advance by one block per iteration. We compare this schedule with individual loads and stores in Table 15.

Fixed-weight sampler The long-term sampler for x and y uses the reference rejection and duplicate checks, so its number of candidate draws can vary. The fixed-work sampler for the ephemeral supports r_1, r_2, and e uses public loop bounds and mask-based duplicate handling [42,44]. Both samplers require the high word of a 32-bit by 32-bit range-conversion product. On Cortex-M3, we compute this word by splitting each operand into 16-bit halves.

$$a = a_{\mathrm{h}}2^{16} + a_{\mathrm{l}}, \qquad b = b_{\mathrm{h}}2^{16} + b_{\mathrm{l}}.$$

We reconstruct the high word from the four 16-bit products $a_{\mathrm{l}}b_{\mathrm{l}}$, $a_{\mathrm{l}}b_{\mathrm{h}}$, $a_{\mathrm{h}}b_{\mathrm{l}}$, and $a_{\mathrm{h}}b_{\mathrm{h}}$, with the middle carry added explicitly. The same routine implements the Barrett step in the long-term sampler and the multiplicative range conversion in

the ephemeral sampler. Neither path therefore depends on the variable-latency Cortex-M3 long multiplier. This timing channel is distinct from the division-instruction channel studied by Schröder et al. [43].

Public-bound vector and decoder kernels We specialized the vector and decoder inner loops with bounds determined by public parameters. Vector addition uses aligned multiple-word transfers. The Reed–Solomon syndrome loop uses the log-domain update described in Section 3.6. The logarithm and exponent tables occupy 1028 bytes; their placement is specified in Section 4.1. Cortex-M3 has no packed halfword addition or subtraction for the 128-point Walsh–Hadamard transform. We unpack two signed 16-bit values from each 32-bit word, compute scalar sums and differences, and pack both results into one stored word.

Cortex-M7 Cortex-M7 provides DSP instructions and DTCM, and its dual-issue pipeline can execute suitable independent instructions within one cycle [5]. We rescheduled the direct accumulator to expose independent operations on Cortex-M7.

Direct accumulator In Direct-Asm on Cortex-M7, we process two words in each main-loop iteration. We use individual `LDR` and `STR` instructions, load both source and accumulator pairs before the shifts and XORs, and compute the next pair's carry after both word updates. Figure 2(b) shows this order. The compact Cortex-M7 implementation uses the four-word Zero-Buffer multiplier, so the two-word schedule described here applies only to Direct-Asm. Table 15 compares the two-word and four-word direct schedules on Cortex-M7.

Sampler and decoder kernels On Cortex-M7, we retain the long-term and ephemeral sampler structures and use the fixed-latency long multiplier to obtain the high word of a native 64-bit product. We use individual loads and stores for vector XOR to expose instruction pairing, and the RS syndrome loop uses the same log-domain update as Cortex-M3. In the 128-point Walsh–Hadamard transform, packed `SADD16` and `SSUB16` instructions compute two signed sums or differences per instruction. We placed the support arrays, accumulator, and stack in DTCM. We placed the field tables in flash for Compact-Asm in the direct–compact cycle comparison and for both HQC-256 regeneration variants (Section 4.1). We measured the sampler and decoder costs separately for the HQC-128 Opt-C and Direct-Asm implementations (Table 17).

3.4 FAFFT ports

We ported the published Cortex-M4 FAFFT implementation to Cortex-M3 and Cortex-M7 without changing its transform structure. Cortex-M3 uses compiler-managed integer state in C and memory-backed scratch instead of floating-point-register scratch in assembly. Cortex-M7 supports the same packed-DSP instructions as Cortex-M4, but its wider pipeline and DTCM can change the benefit of

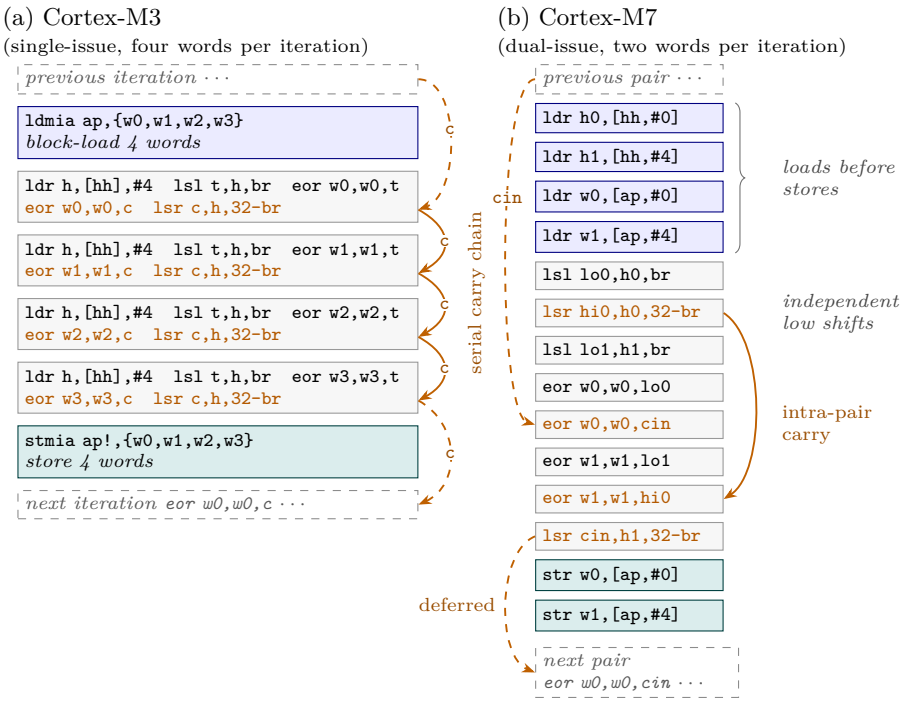

Fig. 2: Accumulator schedules in Direct-Asm: four words per iteration on Cortex-M3 and two on Cortex-M7, shown in program order without issue-slot assignments. Orange arrows show carry dependencies. The pointers `ap` and `hh` address the accumulator and dense operand a; `w0`–`w3` and `h`/`h0`/`h1` hold their words. Here `br`$= \beta$, `t`/`lo0`/`lo1` hold low shifted words, and `hi0` holds the first word's carry. The carry `c` passes between words; `cin` is updated after each pair and passed to the next. The schematic right shift by $32 - \beta$ gives zero when $\beta = 0$.

register scratch. It has neither carry-less multiplication nor vector instructions for the binary ring transform, so tower-field operations remain scalar and the working set is retained. We compare compiler-managed and transferred register strategies on Cortex-M7 (Tables 14 and 16).

3.5 Complete-KEM compact data flow

We access serialized inputs directly and use caller-owned output buffers as temporary workspaces before final serialization, where their size and alignment permit. Figure 3 shows the decapsulation lifetimes; Table 9 also includes decoder, hash, and KEM state omitted from the figure.

Key generation and encapsulation reuse caller-owned outputs as 64-bit-word workspaces. Key generation expands h in the public-key output, which final serialization overwrites after computing s. Encapsulation assigns two aligned

ciphertext-output ranges to u and v and one internal vector to h followed by s. This keeps Zero-Buffer destinations separate from their dense inputs and retains u until serialization. The internal vector is included in the compact encapsulation stack and footprint.

Decapsulation reads the received u and embedded s directly and retains one dense decryption result. Reencryption keeps the temporary h expansion, the r_2 support, and one output workspace. The workspace holds the public-key expansion state, then $u^\star$ and $v^\star$; we compare each component before reuse. The decoded message and rejection key share a buffer at different times.

We retain only r_2 across the reencryption products and regenerate r_1 and e immediately before use by reinitializing the XOF with the same seed and advancing to their positions in the reference sampler's byte stream. This avoids retaining both supports while h and the output workspace coexist. Encapsulation resumes a saved XOF state after the r_2 draw, sharing its storage with the public-key expansion state. Cortex-M3 batches fixed-size XOF requests; Cortex-M7 squeezes incrementally, consuming the same byte stream. Table 18 compares regeneration with drawing r_2, e, and r_1 once in the same order and retaining them across the products, with the multiplication kernels unchanged.

The public-key output for key generation and the ciphertext output for encapsulation require 8-byte alignment for the 64-bit-word workspaces. The ciphertext and secret-key inputs to decapsulation require 4-byte alignment for the direct 32-bit loads. These are preconditions of the evaluated interface, without runtime checks. We evaluated separate input and output objects; the reported footprints exclude copies needed to adapt unaligned caller buffers.

Storage requirements We compare the storage required by the direct and compact HQC-256 decapsulation objects on both cores (Table 6). A padded ring vector occupies $D = 8\lceil n/64 \rceil = 7208$ bytes. The direct ciphertext structure contains two such vectors, including a ring-sized array for the truncated v, and a 16-byte salt. Each received or candidate ciphertext structure therefore occupies $2D + 16 = 14{,}432$ bytes.

We relate these object sizes to the footprint through their lifetimes. Direct-Asm retains both ciphertext structures during reencryption, alongside its operand buffers and multiplication scratch. In Compact-Asm, the D-byte candidate workspace is live only during reencryption, after the decryption result and decoder stack have been released (Figure 3). The support row combines narrower positions with shorter lifetimes: the direct arrays use 32-bit positions, whereas the compact arrays use 16-bit positions. We hold the 16-bit representation fixed in the regeneration comparison (Table 18). The capacities include limb padding but exclude register spills and alignment between stack objects. Because the objects occupy different call frames and phases, their size differences do not sum to a peak-stack reduction.

MUTUALISM shares the storage of h and v: it computes u while h occupies this shared region, then retains u while forming v in the same region [34]. During our reencryption, h and the candidate workspace also coexist for the first product, occupying $2D = 14{,}416$ bytes in HQC-256. Immediate comparison

Table 6: Declared storage for selected HQC-256 decapsulation objects on both cores, in bytes.

Object or phase	Direct-Asm	Compact-Asm	Compact storage and lifetime
Full-product accumulator	14,416	0	Accumulate in the destination.
Parsed received ciphertext	14,432	0	Read the serialized input in place.
Candidate ciphertext / workspace	14,432	7208	Compare $u^\star$, then use the same workspace for $v^\star$.
Reencryption h buffer	7208	7208	Release after the hr_2 product.
Imported s vector	7208	0	Read s from the serialized key.
Ephemeral supports during hr_2	1788	298	Keep only the 16-bit support of r_2 in this phase.
Decryption result vector	7208	7208	Release after decoding, before reencryption.

allows us to use one D-byte writable ring vector for the second product, while reading s from the serialized key. For the first product, the API objects, two ring vectors, and retained r_2 support account for $21{,}786 + 2 \cdot 7208 + 298 = 36{,}500$ bytes. These objects coexist in the present allocation and account for all but 358 and 350 bytes of the measured M3 and M7 footprints, respectively. The sum excludes the remaining call state and alignment. We did not isolate the effect of immediate comparison on the complete-operation peak; the smaller working set of the second product alone does not determine that peak.

The serialized decapsulation API storage remains 21,786 bytes in both implementations. For compact HQC-256 key generation and encapsulation, the 7237-byte public-key output holds the D-byte expansion of h, and the 14,421-byte ciphertext output holds two aligned D-byte workspaces before final serialization. These ranges use API storage already included in the operation footprint.

Equivalence of the compact data flow We justify the lifetime changes for the canonical operands considered in Section 3.2, with the separate input and output objects used in our evaluation. We reuse each region only after its previous value's last use and when the next value meets its size and alignment requirements. We keep each multiplication destination separate from its dense operand and preserve the serialized field boundaries.

We compare each candidate component before reusing its workspace and combine the component differences with bitwise OR. We postpone shared-secret selection until all components have been compared. For the same candidate components, this gives the same mismatch result as comparing the complete ciphertext. We preserve the received salt and the hash input: the contiguous $u \parallel v \parallel \mathit{salt}$ range contains the same bytes as the three fields absorbed in order.

We regenerate each support with the same XOF seed, byte offsets, and reference sampler mapping, reproducing it without retaining it across products. We retain the rejection-mask and shared-secret selection rules. The reduction-boundary discussion in Section 3.2 explains why this argument does not extend to reference-equivalent intermediate values for noncanonical encodings. For a

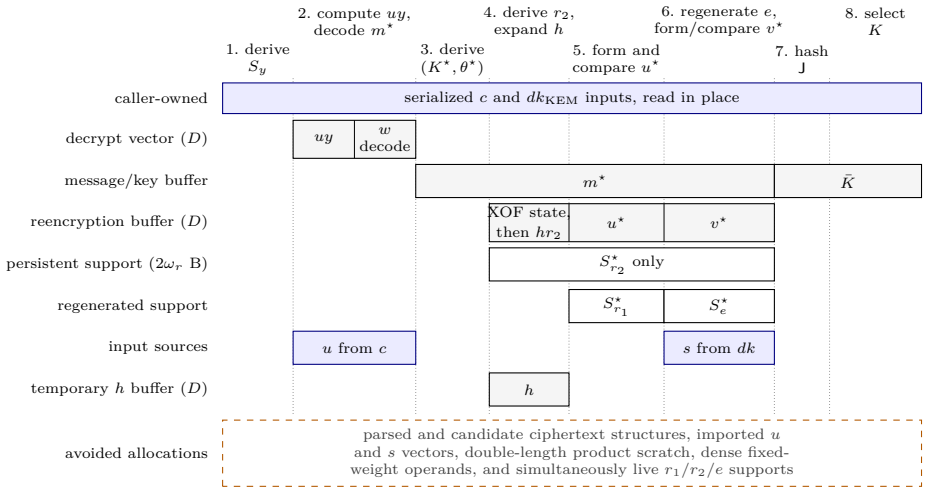

Fig. 3: Compact decapsulation buffer lifetimes on Cortex-M3 and Cortex-M7. Blue boxes mark caller-owned inputs and the fields read from them; gray boxes mark internal buffers, and white boxes mark support lists. The temporary h buffer coexists with the reencryption buffer during the first product. For HQC-256, $D = 7208$ bytes and $2\omega_r = 298$ bytes. "XOF state" denotes the public-key expansion state.

valid secret key, the regenerated $u^\star$ has zero padding, so nonzero padding in the received u necessarily makes the comparison of all serialized bytes fail. We retain this mismatch through the remaining comparisons and select $\mathsf{J}(\mathsf{H}(ek_{\mathrm{KEM}}), \sigma, c)$ using the received ciphertext, independently of the decoded message.

We compared KEM outputs against the reference for the test cases in Section 4.1 and measured memory use as described there. Tables 14 and 16 compare complete-operation cycles for Compact-Asm and Direct-Asm.

3.6 Decoder implementation

As in the reference implementation, our decoder produces a candidate message without a separate decoding-failure indication. We use the reencryption comparison to select the shared secret.

Reed–Solomon arithmetic We replace the reference implementation's carryless field multiplication with log/exp arithmetic. This introduces table accesses whose addresses depend on the operands. For nonzero field elements,

$$ab = \exp\big((\log a + \log b) \bmod 255\big),$$

where log and exp are inverse tables for the nonzero elements of $\mathrm{GF}(2^8)$. We use masks to return zero when either operand is zero. For syndrome row i,

we load the logarithm of each received symbol once and advance the running exponent by $i + 1$ between symbol positions. We reduce each index modulo 255 before accessing the exponent table. This removes one field multiplication from each inner-loop iteration. The numbers of rows and symbols are fixed by public parameters.

Error locator We retained the reference fixed-iteration Berlekamp–Massey (BM) error locator in C for all three parameter sets on Cortex-M3 and Cortex-M7 [36]. These decoder paths contain neither `UDIV` nor `SDIV`. Their public, fixed number of rounds and lack of division instructions remove the division-timing mechanism described by Schröder et al. [43]. In each public round $\mu = 0, \ldots, 2\delta - 1$, we compute the discrepancy d_μ and use masks to select the reference algorithm's coefficient and degree updates. Coefficient updates stop at degree δ. Within the designed correction radius, this locator gives the specified decoding result [17]. Beyond that radius, we execute the same fixed-round state transitions, which neither guarantee successful correction nor provide a new bound on decoding failure. A fixed number of rounds and mask-based updates do not establish branch-free machine code or constant-time execution of the decoder.

4 Evaluation

We compare complete-operation cycles and memory use and measure the effect of support regeneration within the compact design.

4.1 Experimental setup

We evaluated all three HQC parameter sets on Cortex-M3 and Cortex-M7 with GCC 13.2.1, keeping the runtime and platform libraries fixed within each comparison. Table 7 summarizes Opt-C, Direct-Asm, and Compact-Asm; Compact-Asm implements the memory organization in Section 3.5. Table 8 lists the compiler options and locations of the decoder's finite-field logarithm and exponent tables (GF tables). We report memory in bytes and capacities in KiB.

Table 7: Support-form designs used in the evaluation.

Design	Support positions	Product accumulation	Use in evaluation
Opt-C	32-bit	C, full-product buffer	Component profile
Direct-Asm	32-bit	Assembly, full-product buffer	Cycle comparison and component profile
Compact-Asm	16-bit	Assembly, Zero-Buffer	Compact data flow and regeneration comparison

Table 8: Configurations for the complete-operation cycle and memory measurements. GF-table locations are given for Cortex-M3/Cortex-M7.

Measurement	Implementation	HQC parameters	Options; GF tables
Cycle comparison	Direct-Asm	128, 192, 256	`-O3`; SRAM/DTCM
Cycle comparison	Compact-Asm	128, 192, 256	`-O3`; flash/flash
Operation footprint	Compact-Asm	128, 192	`-Os`; SRAM/DTCM
Cycles and footprint	Compact-Asm, regeneration on/off	256	`-O3`; flash/flash
Memory stages	Three designs (M3)	256	`-Os`; SRAM
Accumulator schedules	Direct-Asm	128	`-Os`; SRAM/DTCM

Within each core, we compared Direct-Asm and Compact-Asm with identical board settings, compiler options, and timing protocols. The cycle ratios include differences in GF-table placement as well as kernels and data flow. For the HQC-256 regeneration study, we held table placement fixed and measured both variants within the compact design. For the main Compact-Asm results, we measured HQC-128/192 memory use with `-Os` and cycles with `-O3`; these results describe different configurations, not a single time–memory operating point. We used Direct-Asm as the complete-operation cycle baseline and also compared three cumulative memory organizations for HQC-256 on Cortex-M3 with `-Os` and SRAM-resident GF tables (Table 12). This memory comparison uses separate configurations from the `-O3` cycle and footprint measurements.

Cycle acquisition We ran the Nucleo F207ZG and F767ZI at 30 MHz and 24 MHz, respectively, with zero flash wait states [37]. On Cortex-M7, we enabled the instruction and data caches, flash ART, and prefetch, and ran each operation once immediately before its timed call. We used no warm-up calls on Cortex-M3. For the isolated product measurements in the schedule comparison, we used no explicit warm-up on either core. For timing, we used the core-clocked SysTick counter and its overflow handler, retaining their overhead in the reported counts and emitting output after each timed interval.

For the compact/direct and regeneration comparisons, we timed the KEM call and summarized the first 100 recorded cycle counts for each operation. Before each decapsulation measurement, we generated a fresh key pair and encapsulation, then timed decapsulation of the resulting valid ciphertext. We report medians and use the interquartile range (IQR) to describe within-run variation. The maximum IQR-to-median ratios were 0.283 % in the direct–compact comparison and 0.121 % in the regeneration study. For the FAFFT port comparisons, we used 12 observations on Cortex-M3 and the first 100 on Cortex-M7.

Functional checks We checked the compact implementations against the HQC reference implementation.[4] We used separate software tests to compare two deterministic KEM cases for each parameter set, including the public key, secret key, ciphertext, and encapsulated and decapsulated shared secrets. For each test

[4] `https://gitlab.com/pqc-hqc/hqc`, revision `161cd4f`.

case, we also compared shared secrets after seven ciphertext mutations and two secret-key mutations, covering both ciphertext components, padding, salt, the rejection seed σ, and the embedded public key. The shared-secret output hashes matched across the reference, direct, and compact implementations after direct serialized access and immediate candidate comparison (Section 3.5). For rejected ciphertexts, we compared the selected rejection secrets; we did not compare the decoder's intermediate messages. These comparisons can agree when both implementations reject an input despite different intermediate products, as discussed in Section 3.2.

We compared 4320 decapsulation outputs for ciphertexts with nonzero padding in u against independently computed rejection keys across all three parameter sets and both target architectures, with no mismatches and 144 successful round trips for valid ciphertexts.

On both boards, we checked the HQC-256 regeneration-enabled and regeneration-disabled configurations with ten valid round trips, ten random-secret-key trials, and ten random-ciphertext trials per configuration. The hashes of the shared-secret outputs matched the reference in both configurations. We used separate executables for functional tests, timing, and memory measurements.

Memory accounting We define the operation footprint as the measured peak stack use plus storage for the operation's serialized inputs and outputs: keys, ciphertexts, and shared secrets. We call this input/output storage API storage in the tables and exclude other global and static data from the footprint. For the resource measurements, we used separate public-key, secret-key, and ciphertext objects, aligned to 8 bytes. The output workspaces in Section 3.5 occupy storage within the public-key and ciphertext objects.

We measured stack watermarks with deterministic inputs by filling a stack region with markers and locating the deepest overwritten marker after the operation. On Cortex-M3, we repeated each deterministic test with larger marked regions until an untouched margin remained; on Cortex-M7, we marked the available stack region before a single operation call. We included the measurement program's input-copy operations for encapsulation and decapsulation, but added no separate interrupt-handler or real-time-system reserve. These measurements describe the tested executions and do not establish worst-case stack bounds over all inputs.

For the `-O3` HQC-256 measurements, we used a separate executable to measure memory for each operation under a 40 KiB SRAM limit and measured cycles with separate executables allowing 128 KiB of SRAM. The reported limits identify configurations in which the operations completed; they are not minimum-memory requirements. Table 11 reports the resident data and measured stack use for the decapsulation memory measurements.

Table 9: Compact HQC cycle counts and operation footprints.

Footprint is measured peak stack use plus serialized input/output storage, in bytes; other global and static data are excluded. We used `-O3` and report cycle medians over 100 measurements. We measured HQC-256 with regeneration enabled in Section 4.7, independently of the direct–compact comparisons in Tables 14 and 16. Dashes denote unreported `-O3` footprints; Table 10 gives the `-Os` memory results.

		KeyGen		Encaps		Decaps	
Parameter	Implementation	Footprint	Cycles	Footprint	Cycles	Footprint	Cycles
Primary Cortex-M3 and Cortex-M7 results							
HQC-128	**Compact-Asm, M3**	–	805,616	–	1,753,130	–	3,477,743.5
	Compact-Asm, M7	–	385,220	–	877,087	–	1,835,880.5
HQC-192	**Compact-Asm, M3**	–	1,796,740	–	4,063,117	–	7,164,180
	Compact-Asm, M7	–	934,926	–	2,187,780.5	–	4,054,475
HQC-256	**Compact-Asm, M3**	23,558	3,204,205.5	30,446	7,730,438.5	36,858	13,167,082
	Compact-Asm, M7	23,562	1,764,894	30,438	4,093,131	36,850	7,556,018.5
Published Cortex-M4 context							
HQC-128	MUTUALISM (compact)[a]	9,728	800,000	7,680	1,904,000	12,698	2,909,000
	MUTUALISM (fast)[a]	11,878	597,000	9,933	1,417,000	14,848	2,215,000
HQC-192	MUTUALISM (compact)[a]	18,842	2,080,000	14,643	4,940,000	23,757	7,203,000
	MUTUALISM (fast)[a]	23,347	1,397,000	19,149	3,366,000	28,262	4,939,000
HQC-256	MUTUALISM (compact)[a]	29,798	3,822,000	22,938	9,017,000	37,478	13,193,000
	MUTUALISM (fast)[a]	37,069	2,463,000	30,106	5,903,000	44,749	8,710,000

Bold labels mark the primary Cortex-M3 and Cortex-M7 Compact-Asm results.
[a] MUTUALISM reports operation stack plus per-operation API storage. We converted its Table 2 memory values from KiB to bytes by multiplying by 1024 and rounding. Its cycle counts have a resolution of 10^3 cycles. These published values provide cross-study context rather than matched speedup measurements.

Table 10: Compact HQC-128/192 memory measurements with `-Os`. Stack, API storage, and operation footprint are in bytes; SRAM limits are in KiB.

Parameter	Core	Operation	Stack	API	Footprint	SRAM limit
HQC-128	M3	KeyGen	3748	4562	8310	20
HQC-128	M3	Encaps	3324	6706	10,030	20
HQC-128	M3	Decaps	6476	6786	13,262	20
HQC-128	M7	KeyGen	3744	4562	8306	16
HQC-128	M7	Encaps	3312	6706	10,018	16
HQC-128	M7	Decaps	6084	6786	12,870	16
HQC-192	M3	KeyGen	6164	9116	15,280	24
HQC-192	M3	Encaps	5764	13,524	19,288	24
HQC-192	M3	Decaps	9532	13,612	23,144	28
HQC-192	M7	KeyGen	6160	9116	15,276	28
HQC-192	M7	Encaps	5752	13,524	19,276	28
HQC-192	M7	Decaps	9528	13,612	23,140	28

We compiled each operation with GCC 13.2.1 and `-Os`, and placed the GF tables in SRAM on M3 and DTCM on M7. The SRAM limit is the memory available to each executable. We measured the `-O3` cycles in Table 9 with separate builds.

Table 11: Operation footprints and RAM subtotals for compact HQC-256 decapsulation with `-O3`, in bytes.

Quantity	Cortex-M3	Cortex-M7
Serialized API storage	21,786	21,786
Operation stack watermark	15,072	15,064
Operation footprint	36,858	36,850
Resident data, including API and measurement state	24,224	24,208
Resident data plus operation stack (subtotal)	39,296	39,272
Enforced SRAM limit	40,960	40,960

Table 12: Cumulative HQC-256 decapsulation memory comparison on Cortex-M3 with `-Os` and SRAM-resident GF tables. Stack values and reductions are in bytes; reductions are relative to the preceding row.

Cumulative design	Stack	Reduction	SRAM limit [KiB]
Component-wise comparison with direct accumulation	38,388	–	64
Zero-Buffer accumulation	24,636	13,752	56
Complete compact organization	15,052	9584	40

4.2 Memory footprint and program size

The HQC-256 compact decapsulation footprints are 36,858 bytes on Cortex-M3 and 36,850 bytes on Cortex-M7 (Table 9), including the decoder and hash state omitted from Figure 3. The common serialized API objects, two ring-length vectors, and retained support dominate live storage on both cores (Section 3.5); the measured stacks differ by eight bytes. Table 10 gives the separate `-Os` footprints for HQC-128/192, excluding resident GF tables. Table 11 reports resident data and stack use for the HQC-256 memory executables.

The RAM subtotal is the sum of the measured operation stack and the resident global and static data, which include the API objects. The subtotal excludes alignment padding between sections and stack space used by the calling and measurement code outside the marked region. The operation footprint counts only stack and API storage, which is the metric used in the published MUTUALISM comparison.

Effect of the complete-KEM memory organization We compared three cumulative designs to examine the stack reduction from Zero-Buffer multiplication and the remaining compact transformations (Table 12). We compiled each design with `-Os`, kept the decoder GF tables in SRAM, and used the same operation-specific stack measurement procedure. The serialized API storage is 21,786 bytes in all three configurations. We used the SRAM limits shown in the table; these identify configurations in which decapsulation completed.

The first design already compares ciphertext components as they become available but retains a full-product accumulator. Replacing this accumulator

Table 13: Program sizes for the `-O3` timing measurements: code and read-only data in the `.text` section, in bytes.

Core	Parameter	Direct-Asm	Compact-Asm	Increase
Cortex-M3	HQC-128	53,332	55,892	2560
	HQC-192	54,612	57,436	2824
	HQC-256	55,536	58,360	2824
Cortex-M7	HQC-128	62,796	65,228	2432
	HQC-192	64,268	66,836	2568
	HQC-256	62,148	64,772	2624

with Zero-Buffer multiplication reduced the measured stack by 13,752 bytes. The subsequent compact changes reduced it by another 9584 bytes, a 38.9 % reduction from the Zero-Buffer stage's measured stack use. These changes combine direct access to serialized operands, further lifetime-based reuse, 16-bit support positions, and deterministic support regeneration (Section 3.5). The final 15,052-byte stack belongs to this `-Os`, SRAM-table configuration; Table 11 reports the separate `-O3` configuration with GF tables in flash.

Program size We measured the `.text` section of the executables used for timing in Tables 14 and 16 (Table 13). In these executables, this section contains code and read-only data from the implementation, measurement program, and runtime libraries, together with the interrupt vector table. The compact binaries use 2432–2824 more bytes in this section than Direct-Asm. These differences include the GF tables moved from SRAM/DTCM to flash, so they do not measure an instruction-code increase alone.

4.3 Cortex-M3

On the DSP-free Cortex-M3, the measured compact/direct decapsulation cycle ratio decreases from 1.171× for HQC-128 to 1.081× for HQC-192 and 1.057× for HQC-256 (Table 14). Key generation remains close to the direct design, while encapsulation takes more cycles at every parameter set.

In the separate HQC-128 FAFFT comparison, the assembly port takes 9.7 % more decapsulation cycles than the C port. The port retains the transform structure and working storage described in Section 3.4.

Accumulator scheduling We separately compared the direct accumulator schedules for HQC-128 with `-Os`, holding the other routines and GF-table placement fixed within each core (Table 15). We timed a complete support-form product, including initialization and reduction, and full decapsulation, reporting the medians of the first 100 observations for each measurement. On Cortex-M3, four-word block transfers give a 1.10× product speedup and a 1.04× decapsulation speedup over individual loads and stores. The smaller operation-level gain reflects the sampling, decoding, and hashing work outside the accumulator.

Table 14: Cortex-M3 complete-operation cycle comparison and FAFFT port results.

`-O3` measurements; medians of 100 samples.

Parameter	Operation	Direct-Asm	Compact-Asm	Compact/direct
HQC-128	KeyGen	773,912	805,616	1.041×
	Encaps	1,529,661	1,753,130	1.146×
	Decaps	2,969,428	3,477,743.5	1.171×
HQC-192	KeyGen	1,719,255.5	1,796,740	1.045×
	Encaps	3,751,908.5	4,063,117	1.083×
	Decaps	6,629,265	7,164,180	1.081×
HQC-256	KeyGen	3,232,860	3,204,207.5	0.991×
	Encaps	7,119,021	7,730,440.5	1.086×
	Decaps	12,460,380.5	13,167,078	1.057×

HQC-128 FAFFT ports, measured separately with `-O3`, $N = 12$.

Implementation	Decapsulation cycles
FAFFT C port	11,742,090.5
FAFFT assembly port	12,876,099

Table 15: HQC-128 direct-accumulator schedule comparisons with `-Os`. Values are median cycles from the first 100 observations.

Core	Direct accumulator schedule	Product cycles	Decaps cycles
Cortex-M3	Four-word, individual `LDR/STR`	416,669	3,168,249
	Four-word, `LDMIA/STMIA`	380,369	3,049,452
Cortex-M7	Four-word, `LDMIA/STMIA`	202,367	1,620,318.5
	Two-word, individual `LDR/STR`	188,589	1,574,075

4.4 Cortex-M7

For the evaluated Cortex-M7 configurations, the compact/direct decapsulation cycle overhead decreases with the parameter set (Table 16). It exceeds the encapsulation overhead, which in turn exceeds the key-generation overhead. For HQC-256, the decapsulation increase is 14.2 %, compared with 5.7 % on Cortex-M3; their compact footprints are nearly equal.

In the separate `-Os` schedule comparison, two-word accumulation gives a 1.07× product speedup and a 1.03× decapsulation speedup over the four-word schedule (Table 15). Direct-Asm uses this two-word schedule; Compact-Asm uses the four-word Zero-Buffer schedule in Section 3.3.

In the separate HQC-128 FAFFT comparison, the assembly port takes 5.34 % fewer decapsulation cycles than the C port. The two port comparisons thus give opposite C-to-assembly orderings on the evaluated targets, with a benefit from the transferred assembly kernels on Cortex-M7 and an additional cost on Cortex-M3.

Table 16: Cortex-M7 complete-operation cycle comparison and FAFFT port results.

`-O3` measurements; medians of 100 samples.

Parameter	Operation	Direct-Asm	Compact-Asm	Compact/direct
HQC-128	KeyGen	368,581.5	385,220	1.045×
	Encaps	766,943.5	877,087	1.144×
	Decaps	1,548,442.5	1,835,880.5	1.186×
HQC-192	KeyGen	888,395	934,926	1.052×
	Encaps	1,947,435.5	2,187,780.5	1.123×
	Decaps	3,486,554.5	4,054,475	1.163×
HQC-256	KeyGen	1,671,671	1,764,819	1.056×
	Encaps	3,738,322.5	4,091,588	1.094×
	Decaps	6,615,724	7,554,869	1.142×

HQC-128 FAFFT ports, measured separately with `-O3`, $N = 100$.

Implementation	Decapsulation cycles
FAFFT C port	5,746,323.5
FAFFT assembly port	5,439,682.5

Table 17: Separate Cortex-M7 HQC-128 decapsulation profile with `-O3`, in cycles.

Component	Opt-C	Direct-Asm	C/assembly
Support generation	585,789	119,483.5	4.90×
Support-form products	1,150,624.5	675,403.5	1.70×
Code decoding	802,988.5	354,980	2.26×
KEM hashes and public-key expansion	422,764	422,538.5	1.00×
Copying and other work	102,351.5	36,422.5	2.81×
Complete decapsulation	3,064,517.5	1,608,828	1.90×

Component profile We profiled Opt-C and Direct-Asm for HQC-128 decapsulation on Cortex-M7 in a separate `-O3` experiment (Table 17). We used 100 repetitions of one fixed valid ciphertext after eight warm-up calls. For each component, we summed the medians of its counters. Support generation includes sampler XOF work, products include initialization and reduction, and decoding includes RM and RS. The hash/expansion row includes public-key expansion and parsing. We assigned copying and work outside the listed categories to the residual, calculated from the total and component counters. The rows therefore sum to the total by construction.

Products, support generation, and decoding take fewer measured cycles with the assembly implementation, while hash/expansion costs remain similar. The reductions across these components explain the complete-operation improvement in this profile: optimizing the support-form product addresses only one part of decapsulation.

4.5 Comparison with Cortex-M4 implementations

Table 9 compares our results with the published compact and fast Cortex-M4 implementations from MUTUALISM [34]. Those results use the earlier Round-4 specification; our implementation follows the August 2025 specification. Across the three parameter sets, our reported KeyGen footprints are smaller than MUTUALISM's compact figures, while MUTUALISM reports smaller Encaps footprints throughout. Its HQC-128 Decaps footprint is also smaller; our HQC-192 and HQC-256 Decaps footprints are smaller (Tables 9 and 10). Our encapsulation retains an internal dense operand in addition to the ciphertext workspace, whereas decapsulation reads s from the key and reuses the candidate workspace after each comparison (Section 3.5). Differences in specification, core, build, and measurement setup prevent attributing the cross-study footprint gaps to individual transformations or interpreting cycle ratios as matched speedups.

4.6 Timing diagnostics

For compact HQC-128 decapsulation on Cortex-M3, we used `-O3` with GF tables in SRAM. With one fixed key pair, we pseudorandomly interleaved fresh valid encapsulations and fresh pseudorandom byte strings of ciphertext length, obtaining 3026 valid and 2974 invalid observations. Using all completed observations, Welch's two-sample t-test gave an absolute statistic of $|t| = 0.15$, below the conventional threshold of 4.5 [18,40].

For positive controls, we inserted delays after decapsulation for the invalid class, within the timed interval, and collected 700 observations in total across both classes per setting. With no injected delay, $|t| = 0.64$; delays producing invalid-minus-valid mean differences of 36.1 and 166.1 cycles gave $|t| = 5.64$ and 25.79, respectively. This fixed-key grouping does not isolate timing dependence on decoded messages or reencryption supports (Section 2.4), nor compare different secret keys.

4.7 Effect of deterministic XOF regeneration

We isolated regeneration for HQC-256 to measure how much complete-operation stack use changes when ephemeral supports are retained for shorter lifetimes. With regeneration disabled, we sampled r_2, e, and r_1 once in the specification-defined order and retained all three supports across the products. With regeneration enabled, we retained r_2 and re-derived later supports from the same positions in the XOF stream when needed. We held the Zero-Buffer kernels, serialized-input access, output reuse, and remaining compact transformations fixed.

We measured stack use under a 40 KiB SRAM limit and cycles with separate executables allowing 128 KiB, using `-O3` for both regeneration configurations (Table 18). In addition to the functional tests in Section 4.1, we checked the reference shared secret in each decapsulation executable used for memory measurement. We evaluated encapsulation and decapsulation because regeneration changes these operations.

Table 18: HQC-256 regeneration comparison within the compact design, with `-O3`.

Core	Operation	Regeneration	Stack [B]	Cycles
M3	Encaps	Disabled	9060	7,550,505
		Enabled	8756	7,730,438.5
	Decaps	Disabled	15,660	12,875,414.5
		Enabled	15,072	13,167,082
M7	Encaps	Disabled	9052	3,980,723
		Enabled	8748	4,093,131
	Decaps	Disabled	15,664	6,973,743.5
		Enabled	15,064	7,556,018.5

Regeneration reduces measured encapsulation stack use by 304 bytes on each core, with cycle increases of 2.38 % on Cortex-M3 and 2.82 % on Cortex-M7. We retain only r_2 during the hr_2 product in decapsulation and regenerate r_1 and e before their subsequent use (Figure 3). This reduces the retained 16-bit support arrays from $3 \cdot 149 \cdot 2 = 894$ bytes to $149 \cdot 2 = 298$ bytes, a difference of 596 bytes in this phase. With the other compact transformations held fixed, we measured complete-operation stack reductions of 588 bytes on Cortex-M3 and 600 bytes on Cortex-M7, with cycle increases of 2.27 % and 8.35 %, respectively. The measured stack reductions are close to the storage released by the two supports.

The similar stack savings on the two cores come at a higher relative cycle cost on Cortex-M7 under these configurations. As described in Section 3.5, encapsulation resumes a saved XOF state, whereas decapsulation restarts the stream and advances to the required supports. We did not measure the costs of these steps separately. Both regeneration configurations completed under the tested 40 KiB memory limit. Retaining the supports provides the lower-cycle choice in our timing measurements, while regeneration makes more stack space available at the measured execution-time cost.

5 Conclusion

We present complete implementations of all three HQC parameter sets on Cortex-M3 and Cortex-M7. We organize the compact design around the lifetimes of values across the complete KEM: direct serialized-input access avoids parsed copies, component-wise ciphertext comparison permits workspace reuse, and deterministic support regeneration replaces retained state with recomputation. We combine these choices with Zero-Buffer multiplication to avoid a full-product buffer. In the cumulative Cortex-M3 HQC-256 comparison with `-Os` and SRAM-resident decoder tables, the remaining representation and lifetime changes reduced the decapsulation stack by 9584 bytes, or 38.9 % relative to the configuration already using Zero-Buffer multiplication and component-wise ciphertext comparison. For HQC-256, the objects that coexist during reencryption account for most of the measured decapsulation footprint.

For HQC-256 decapsulation compiled with `-O3`, we measured operation footprints of 36,858 bytes on Cortex-M3 and 36,850 bytes on Cortex-M7. Within this design, regenerating ephemeral supports saved 588 and 600 bytes of stack with cycle increases of 2.27 % and 8.35 %, respectively. On both cores, decapsulation completed under the tested 40 KiB SRAM limit with and without support regeneration. Retaining supports offers the lower-cycle choice in our timing measurements; regeneration leaves more stack space for the application.

The implementations retain support-dependent control flow and memory accesses. Future work includes regularizing memory accesses throughout the KEM, adding masking, and evaluating physical leakage. Existing masked HQC designs [19] provide implementation baselines, while attacks on specific masking approaches identify leakage mechanisms to evaluate [27].

References

1. Aissaoui, R.: Assessment and Optimization of Post-Quantum Cryptographic Protocols for Civil UAV Communications. Ph.D. thesis, Université de Toulouse, École nationale de l'aviation civile (ENAC) (Dec 2024), https://enac.hal.science/tel-05000998, hAL thesis tel-05000998; defended Dec. 4, 2024
2. Aissaoui, R., Deneuville, J.C., Guerber, C., Pirovano, A.: A performant quantum-resistant KEM for constrained hardware: Optimized HQC. In: Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024). pp. 668–673. SCITEPRESS (2024). https://doi.org/10.5220/0012757800003767, https://www.scitepress.org/Papers/2024/127578/127578.pdf
3. Arm Limited: Cortex-M3 Technical Reference Manual. Arm Limited (2010), https://www.keil.com/dd/docs/datashts/arm/cortex_m3/r2p1/ddi0337i_cortexm3_r2p1_trm.pdf, arm DDI 0337I, revision r2p1
4. Arm Limited: ARM Cortex-M4 Processor Technical Reference Manual. Arm Limited (2013), https://else.fcim.utm.md/pluginfile.php/48982/mod_resource/content/1/ARM%20Cortex-M4%20Processor.pdf, arm DDI 0439D, revision r0p1
5. Arm Limited: Arm Cortex-M7 Processor Technical Reference Manual. Arm Limited (2018), https://developer.arm.com/documentation/ddi0489/f, arm DDI 0489F, revision r1p2
6. Banegas, G., Smith, B., Zahreddine, J.: Exploiting load/store leakage of sparse vectors for key recovery in HQC. Cryptology ePrint Archive, Paper 2026/1491 (2026), https://eprint.iacr.org/2026/1491
7. Botros, L., Kannwischer, M.J., Schwabe, P.: Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Progress in Cryptology – AFRICACRYPT 2019. LNCS, vol. 11627, pp. 209–228. Springer (2019). https://doi.org/10.1007/978-3-030-23696-0_11
8. Botros, L., Kannwischer, M.J., Schwabe, P.: nttm4: Kyber implementation artifact. Software artifact for Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4 (2019), https://github.com/mupq/nttm4/tree/6bbee0eff7a0b9a0fbf1ff158f59c49e024d76b3, the m4round2 implementation compares reencryption components as they are produced
9. Chen, M.S., Chien, T.Y., Chiu, C.M., Huang, C., Lin, H.H., Peng, C.T., Yang, B.Y.: Strided Frobenius additive FFT and its application to HQC. Cryptology ePrint Archive, Paper 2026/1588 (2026), https://eprint.iacr.org/2026/1588

10. Chen, M.S., Chien, T.Y., Chiu, C.M., Lin, H.H., Peng, C.T., Yang, B.Y.: Additive FFTs for HQC on ARM Cortex-M4, revisited. IACR Transactions on Cryptographic Hardware and Embedded Systems **2026**(3), 377–401 (2026). https://doi.org/10.46586/tches.v2026.i3.377-401, https://eprint.iacr.org/2026/739
11. Chen, M.S., Chou, T.: Classic McEliece on the ARM Cortex-M4. IACR Trans. Cryptogr. Hardw. Embed. Syst. (TCHES) **2021**(3), 125–148 (2021). https://doi.org/10.46586/tches.v2021.i3.125-148
12. Chen, M.S., Chou, T., Krausz, M.: Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. IACR Trans. Cryptogr. Hardw. Embed. Syst. (TCHES) **2021**(3), 97–124 (2021). https://doi.org/10.46586/tches.v2021.i3.97-124
13. Chou, T.: QcBits: Constant-time small-key code-based cryptography. In: Cryptographic Hardware and Embedded Systems – CHES 2016. LNCS, vol. 9813, pp. 280–300. Springer (2016). https://doi.org/10.1007/978-3-662-53140-2_14
14. Dong, B., Feng, H., Wang, Q.: OptHQC: Optimize HQC for high-performance post-quantum cryptography. In: 2026 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 410–414. IEEE (2026). https://doi.org/10.1109/ISCAS66217.2026.11562441
15. Dong, H., Guo, Q.: Breaking optimized HQC: The first cache-timing full decryption oracle key-recovery attack in post-quantum cryptography. In: Advances in Cryptology – CRYPTO 2026. LNCS, vol. 16803, pp. 224–255. Springer (2026). https://doi.org/10.1007/978-3-032-35398-6_8, https://eprint.iacr.org/2026/693
16. Dong, H., Guo, Q., Nabokov, D.: Single-trace key recovery attacks on HQC using valid and invalid ciphertexts. In: Advances in Cryptology – EUROCRYPT 2026. LNCS, vol. 16547, pp. 568–597. Springer (2026). https://doi.org/10.1007/978-3-032-25336-1_20, https://eprint.iacr.org/2025/1987
17. Gaborit, P., Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Persichetti, E., Zémor, G., Bos, J., Dion, A., Lacan, J., Robert, J.M., Véron, P., Barreto, P.L., Ghosh, S., Gueron, S., Güneysu, T., Misoczki, R., Richter-Brockmann, J., Sendrier, N., Tillich, J.P., Vasseur, V.: Hamming quasi-cyclic (HQC). Algorithm specifications and supporting documentation (2025), https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf, specification dated August 22, 2025
18. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side-channel resistance validation. In: NIST Non-Invasive Attack Testing Workshop (NIAT) (2011), https://csrc.nist.gov/CSRC/media/Events/Non-Invasive-Attack-Testing-Workshop/documents/08_Goodwill.pdf
19. Goy, G., Spyropoulos, M., Aragon, N., Gaborit, P., Pacalet, R., Perion, F., Sauvage, L., Vigilant, D.: Side-channel sensitivity analysis on HQC: Towards a fully masked implementation. Cryptology ePrint Archive, Paper 2025/1344 (2025), https://eprint.iacr.org/2025/1344
20. Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. IACR Trans. Cryptogr. Hardw. Embed. Syst. (TCHES) **2022**(3), 223–263 (2022). https://doi.org/10.46586/tches.v2022.i3.223-263
21. Hesse, D., Krausz, M., Murugananthan, R., Wollinger, T., Güneysu, T.: Power reveals timing conceals - side-channel attacks and hiding countermeasures for HQC's fixed-weight vector sampling. Cryptology ePrint Archive, Paper 2026/1462 (2026), https://eprint.iacr.org/2026/1462

22. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Theory of Cryptography (TCC 2017). LNCS, vol. 10677, pp. 341–371. Springer (2017). https://doi.org/10.1007/978-3-319-70500-2_12
23. HQC Team: Hamming quasi-cyclic (HQC) (Oct 2026), https://pqc-hqc.org/doc/hqc_specifications_2026_10_12.pdf, specification, October 12, 2026
24. Huang, S., Sim, R.Q., Chuengsatiansup, C., Guo, Q., Johansson, T.: Cache-timing attack against HQC. IACR Trans. Cryptogr. Hardw. Embed. Syst. (TCHES) **2023**(3), 136–163 (2023). https://doi.org/10.46586/tches.v2023.i3.136-163, https://tches.iacr.org/index.php/TCHES/article/view/10959
25. Jang, J., Lee, M., Kwon, D., Hong, S., Kim, S., Lee, S.: Efficient polynomial multiplication for HQC on ARM Cortex-M4. IACR Transactions on Cryptographic Hardware and Embedded Systems **2026**(4), 998–1021 (2026). https://doi.org/10.46586/tches.v2026.i4.998-1021, https://eprint.iacr.org/2025/1939
26. Jang, J., Shin, H., Kim, S., Hong, S., Kwon, D.: Optimizing polynomial multiplication and fixed-weight sampling for HQC on ARM Cortex-M4. Cryptology ePrint Archive, Paper 2026/1450 (2026), https://eprint.iacr.org/2026/1450
27. Jeon, J., Park, Y., Lee, J., Lee, S., Kim, D., Kim, Y.S.: Codeword masking can be harmful under replay attacks on HQC. Cryptology ePrint Archive, Paper 2026/071 (2026), https://eprint.iacr.org/2026/071, ASIACRYPT 2026; revised September 17, 2026
28. Kannwischer, M.J., Petri, R.: pqm4: NISTPQC round 3 results on the Cortex-M4. Third NIST PQC Standardization Conference (2021), https://csrc.nist.gov/presentations/2021/pqm4-nistpqc-round-3-results-on-the-cortex-m4
29. Kim, D., Choi, J., Yoon, S., Seo, S.C.: Optimized implementation of HQC on Cortex-M4. ICT Express **11**(5), 939–944 (2025). https://doi.org/10.1016/j.icte.2025.07.001
30. Lablanche, J., Mortajine, L., Benchaalal, O., Cayrel, P.L., Mrabet, N.E.: Optimized implementation of the NIST PQC submission ROLLO on microcontroller. Cryptology ePrint Archive, Paper 2019/787 (2019), https://eprint.iacr.org/2019/787
31. Lai, Z., Zhang, R., Zhang, Z., Hermelink, J., Schwarz, M., Pham, V.T., Parampalli, U.: You only decapsulate once: Ciphertext-independent single-trace passive side-channel attacks on HQC. Cryptology ePrint Archive, Paper 2025/2162 (2025), https://eprint.iacr.org/2025/2162
32. Lee, M., Jang, J., Kim, S., Hong, S.: Improved Frobenius FFT for code-based cryptography on Cortex-M4. IEEE Internet of Things Journal **12**(17), 36318–36327 (2025). https://doi.org/10.1109/JIOT.2025.3583021
33. Li, W.D., Chen, M.S., Kuo, P.C., Cheng, C.M., Yang, B.Y.: Frobenius additive fast Fourier transform. In: Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation. pp. 263–270. ISSAC '18, ACM (2018). https://doi.org/10.1145/3208976.3208998
34. Liu, C., Maruko, A., Takahashi, Y., Yanai, N.: MUTUALISM: Low-footprint and high-throughput software implementation of HQC for resource-constrained devices. In: Proceedings of the 19th ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 205–216 (2026). https://doi.org/10.1145/3765613.3811678
35. Maillet, N., Nugier, C., Migliore, V., Deneuville, J.C.: Key recovery from side-channel power analysis attacks on non-SIMD HQC decryption. In: Advances in Cryptology – CRYPTO 2025. LNCS, vol. 16004, pp. 70–102. Springer (2025). https://doi.org/10.1007/978-3-032-01901-1_3, https://eprint.iacr.org/2025/1270

36. Massey, J.L.: Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory **15**(1), 122–127 (1969). https://doi.org/10.1109/TIT.1969.1054260
37. MUPQ Team: pqm3: Post-quantum crypto library for the ARM Cortex-M3. GitHub repository (2023), https://github.com/mupq/pqm3
38. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. NIST FIPS 203 (2024). https://doi.org/10.6028/NIST.FIPS.203
39. National Institute of Standards and Technology: NIST selects HQC as fifth algorithm for post-quantum encryption. NIST news release (2025), https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption
40. Reparaz, O., Balasch, J., Verbauwhede, I.: Dude, is my code constant time? In: Design, Automation & Test in Europe (DATE) 2017. pp. 1697–1702. IEEE (2017). https://doi.org/10.23919/DATE.2017.7927267
41. Schamberger, T., Holzbaur, L., Renner, J., Wachter-Zeh, A., Sigl, G.: A power side-channel attack on the Reed-Muller Reed-Solomon version of the HQC cryptosystem. In: Post-Quantum Cryptography (PQCrypto 2022). LNCS, vol. 13512, pp. 327–352. Springer (2022). https://doi.org/10.1007/978-3-031-17234-2_16
42. Schöffel, M., Feldmann, J., Wehn, N.: Efficient hardware implementation of constant time sampling for HQC. arXiv:2309.16493 (2023), https://arxiv.org/abs/2309.16493
43. Schröder, R.L., Gast, S., Guo, Q.: Divide and surrender: Exploiting variable division instruction timing in HQC key recovery attacks. In: 33rd USENIX Security Symposium (USENIX Security 2024). pp. 6669–6686. USENIX Association (2024), https://www.usenix.org/conference/usenixsecurity24/presentation/schr%C3%B6der
44. Sendrier, N.: Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Paper 2021/1631 (2021), https://eprint.iacr.org/2021/1631
45. Šrubař, V.: Implementing Post-Quantum Cryptography on Constrained Devices. Bachelor's thesis, Brno University of Technology, Faculty of Electrical Engineering and Communication (2026), https://www.vut.cz/en/students/final-thesis/detail/175178?zp_id=175178
46. STMicroelectronics: RM0410: STM32F76xxx and STM32F77xxx Advanced Arm-Based 32-Bit MCUs. STMicroelectronics (2024), https://www.st.com/resource/en/reference_manual/rm0410-stm32f76xxx-and-stm32f77xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf, reference manual, Rev. 5, July 2024
47. STMicroelectronics: RM0033: STM32F205xx, STM32F207xx, STM32F215xx, and STM32F217xx Advanced Arm-Based 32-Bit MCUs. STMicroelectronics (2025), https://www.st.com/resource/en/reference_manual/rm0033-stm32f205xx-stm32f207xx-stm32f215xx-and-stm32f217xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf, reference manual, Rev. 10, May 2025
48. Tasopoulos, G., Li, J., Fournaris, A.P., Zhao, R.K., Sakzad, A., Steinfeld, R.: Performance evaluation of post-quantum TLS 1.3 on resource-constrained embedded systems. In: Information Security Practice and Experience. LNCS, vol. 13620, pp. 432–451. Springer (2022). https://doi.org/10.1007/978-3-031-21280-2_24
49. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: A generic power/EM analysis on post-quantum KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. (TCHES) **2022**(1), 296–322 (2022). https://doi.org/10.46586/tches.v2022.i1.296-322

50. Urian, R.: Classic McEliece on ARM M4. NIST pqc-forum post (Apr 2021), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/PHn7rjmZdfs, follow-up post, April 2, 2021
51. Wafo-Tapa, G., Bettaieb, S., Bidoux, L., Gaborit, P., Marcatel, E.: A practicable timing attack against HQC and its countermeasure. Advances in Mathematics of Communications **16**(3), 621–642 (2022). https://doi.org/10.3934/amc.2020126, https://eprint.iacr.org/2019/909