Fast Implementation of Binary Field Multiplication on Arm Helium with Applications to Post-Quantum Signatures

Eric Azevedo de Oliveira[1], **Décio Luiz Gazzoni Filho**[2,3], **Felix Carvalho Rodrigues**[1,4], **Julio López**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

[2]Technology Innovation Institute (TII), Abu Dhabi, UAE

[3]Departamento de Engenharia Elétrica – Universidade Estadual de Londrina

[4]Universidade Virtual do Estado de São Paulo (UNIVESP)

e291073@dac.unicamp.br,
{felix.rodrigues,jlopez}@ic.unicamp.br,
decio.gazzoni@tii.ae

Abstract. *Polynomial multiplication over binary fields* $\mathrm{GF}(2^n)$ *is a major bottleneck in post-quantum cryptographic algorithms, demanding constant-time implementations on constrained devices. This paper presents constant-time algorithms for multiplication and squaring in* $\mathrm{GF}(2^n)$*, with* $n \in \{128, 192, 256\}$*, tailored for the Arm Cortex-M85 and applied to the AIMer and FAEST signature schemes. By leveraging the* `VMULLB` *and* `VMULLT` *carry-less multiplication instructions from the Helium MVE extension, we deliver vectorized routines integrated into the reference codebases of both schemes. Benchmarks on the Renesas EK-RA8M1 board show speedups of up to* $1.35\times$ *for AIMer and* $1.30\times$ *for FAEST-EM.*

1. Introduction

Polynomial multiplication over binary extension fields is a central operation in many cryptographic algorithms. In symmetric cryptography, it appears in AES, the most widely used block cipher in practice. The growing family of post-quantum signature schemes whose security is based on symmetric primitives also relies heavily on efficient field arithmetic. Two representative examples are FAEST [Baum et al. 2025] and AIMer [Ha et al. 2024]. FAEST, a candidate currently in the second round of the NIST post-quantum digital signature standardization process [Alagic et al. 2024], is based on the VOLE-in-the-Head (VOLEitH) paradigm, which treats signature generation as a non-interactive zero-knowledge (NIZK) argument that the signer knows a secret AES key corresponding to a public key. Its security is directly tied to AES-128, AES-192, and AES-256, requiring multiplications in $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$ during key generation, signing, and verification. AIMer, recently standardized by the Korean KpqC [Firmin 2025] competition, is built around the tweakable one-way function AIM2, whose nonlinear layer consists of parallel inverse Mersenne S-boxes performing exponentiations of the form x^{2^e-1} over the same three fields; these decompose into sequences of squarings and field multiplications [Kim et al. 2023] that are invoked repeatedly on every evaluation of AIM2.

Efficient deployment of these schemes on resource-constrained devices requires fast binary polynomial multiplication, since this operation is invoked many times per signature operation. In this setting, non-vectorized implementations become a performance

bottleneck. Constant-time execution is also required, as data-dependent control flow exposes implementations to side-channel attacks. Recent Arm Cortex-M processors, such as the Cortex-M85, include the M-Profile Vector Extension (MVE), also referred to as Helium, which provides eight 128-bit SIMD registers and carry-less polynomial multiplication instructions (VMULLB and VMULLT). These features make the Cortex-M85 a suitable platform for vectorized and constant-time binary field arithmetic on embedded devices, and motivate the work presented here.

In this work, we present fully vectorized and constant-time algorithms for multiplication and squaring in $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, targeting the Helium instruction set on the Arm Cortex-M85. The proposed multiplication algorithm exploits the even/odd lane decomposition of the VMULLB and VMULLT instructions through a sliding-window accumulation scheme. We further show that squaring can be performed using register rearrangement (permutation) and carry-less multiplication instructions, making it faster than general multiplication. The proposed primitives are integrated into FAEST and AIMer, and their performance is evaluated on a Renesas EK-RA8M1 board equipped with a Cortex-M85 processor.

Related Work. Efficient implementations of multiplication in binary fields have been extensively studied in the literature. [López and Dahab 2000] introduced an efficient algorithm for multiplication in $\mathrm{GF}(2^m)$ using polynomial basis representation, employing look-up tables to achieve speedups of up to $5.5\times$ over the standard shift-and-add method. Subsequent works explored optimizations at the implementation level. In particular, [Oliveira et al. 2014] proposed a multiplier for 64-bit binary polynomials using SIMD instructions from the Arm architecture, combined with Karatsuba decomposition to scale to larger operands, demonstrating significant performance gains.

Furthermore, efficient techniques for arithmetic in $GF(2^m)$, including decomposition and operation reorganization strategies that reduce the complexity of polynomial multiplication, have been investigated in works such as [Weimerskirch et al. 2003]. Although the focus of these works remains at the algorithmic level, the proposed approaches are relevant for efficient implementations because they directly influence how operations are mapped to low-level instructions. More recently, [Becker et al. 2021] investigated high-degree polynomial multiplication for post-quantum schemes on the Cortex-M4 and Cortex-M55, exploring strategies based on decompositions such as Toom–Cook and Karatsuba, as well as optimizations related to efficient register usage and memory access. Their work targets multiplication of polynomials with integer coefficients modulo 16- or 32-bit moduli, rather than binary polynomial multiplication over $\mathbb{F}_2[x]$. Although their focus is on lattice-based schemes, their analysis is relevant for efficient implementations on Arm architectures.

Regarding signature schemes, [Kannwischer et al. 2019] provided a comprehensive evaluation of additional NIST candidates on the Cortex-M4, including an initial benchmark for AIMer on that platform. For FAEST, [Aranha et al. 2025] presented a compact, constant-time implementation based on a VOLE abstraction, reducing memory usage and reporting latencies. Our work targets the Cortex-M85, exploring the Helium Extension (MVE), and focuses on optimizing multiplications and squarings in binary fields, which constitute the dominant arithmetic cost in schemes such as AIMer and FAEST.

Our Contributions. The main contributions of this work are summarized as follows:

- We present constant-time implementations of multiplication and squaring in $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$ for the Arm Cortex-M85. By leveraging the carry-less polynomial multiplication instructions, we accelerate the FAEST and AIMer signature schemes.
- We evaluate the proposed field arithmetic in the context of AIMer and FAEST on a Renesas EK-RA8M1 (Cortex-M85) board, reporting competitive performance over the reference implementations.

Paper Structure. The remainder of this paper is organized as follows. Section 2 introduces binary field arithmetic and the polynomial basis representation used throughout this work. Section 3 describes the Arm Helium MVE instruction set, with emphasis on the carry-less multiplication instructions `VMULLB` and `VMULLT`. Section 4 presents the proposed multiplication, squaring, and modular reduction algorithms for $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$. Section 5 describes the integration of the proposed implementations into the AIMer and FAEST signature schemes. Section 6 reports the experimental results on the Renesas EK-RA8M1 board.

2. Binary Field Arithmetic

Let $\mathrm{GF}(2^m)$ be a binary extension field. In a polynomial-basis representation, each element $a \in \mathrm{GF}(2^m)$ is written as a polynomial $a(x) = a_0 + a_1 x + \cdots + a_{m-1}x^{m-1}$ over $\mathrm{GF}(2)$ and stored as the binary vector $(a_0, a_1, \ldots, a_{m-1})$ of length m, packed into $t = \lceil m/W \rceil$ words of width W. Multiplication of two elements $a, b \in \mathrm{GF}(2^m)$ proceeds in two stages. The first computes the carry-less polynomial product $c(x) = a(x)\,b(x)$, which has degree at most $2m - 2$. A straightforward approach iterates through the bits of one operand and, for each bit, conditionally XORs a shifted copy of the other operand into an accumulator, requiring m shift and XOR operations per multiplication. The second stage reduces $c(x)$ modulo a degree-m irreducible polynomial $f(x) = x^m + r(x)$ to obtain an element of $\mathrm{GF}(2^m)$. In the applications discussed here, the polynomial $r(x)$ has low degree, specifically $deg\, r(x) < 8$, which allows the modular reduction to be carried out efficiently with a small number of carry-less multiplications.

3. ARM Helium MVE

Helium is the SIMD extension introduced in the Armv8.1-M architecture, implemented on the Cortex-M55 and Cortex-M85 processors [Marsh 2020, Arm Ltd. 2023]. It is formally known as the M-Profile Vector Extension (MVE), though throughout this work we use the commercial name Helium. It adds eight 128-bit vector registers (Q0–Q7) processed by a dedicated Extended Processing Unit (EPU) that operates independently from the scalar integer pipeline. Each register can be viewed as 16 lanes of 8-bit elements, 8 lanes of 16 bits, 4 lanes of 32 bits, or 2 lanes of 64 bits. Vector instructions execute in a beat-based microarchitecture where each 128-bit operation is divided into two beats, each processing half of the register per clock cycle; consecutive instructions can overlap in the EPU pipeline, with one instruction executing its second beat while the next is in its first.

For binary field arithmetic, the relevant instructions are the carry-less polynomial multiply instructions `VMULLB.P8`, `VMULLT.P8`, `VMULLB.P16`, and `VMULLT.P16`.

Given two registers A and B, `VMULLB` takes the even-indexed elements and computes their pairwise polynomial products, while `VMULLT` operates on the odd-indexed elements, both producing widened results: for `.P8`, eight 8-bit inputs yield eight 16-bit products per register, and for `.P16`, four 16-bit inputs yield four 32-bit products. Together, `VMULLB` and `VMULLT` cover all lanes of a 128-bit register in two instructions, computing the complete set of carry-less products across the full register width. Figure 1 illustrates `VMULLB.P16` and `VMULLT.P16`.

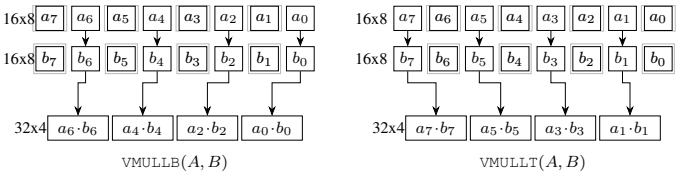

Figure 1. Operation between polynomial multiplication instructions

4. Specialized multiplication

This section describes the proposed algorithms for carry-less polynomial multiplication and squaring in $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, designed to exploit the Helium instruction.

4.1. 128-bit direct multiplication

Let $a, b \in \mathrm{GF}(2^{128})$. Write a as $a(x) = \sum_{i=0}^{7} a_i x^{16i}$ and $b(x) = \sum_{j=0}^{7} b_j x^{16j}$, where each a_i, b_j is a 16-bit word representing a polynomial of degree < 16 over $\mathrm{GF}(2)$. The carry-less product expands as $a(x) \cdot b(x) = \sum_{k=0}^{14} \left(\sum_{i+j=k} a_i b_j \right) x^{16k}$. This process is shown in Algorithm 1.

Splitting even and odd lanes. The eight coefficients of a and b naturally partition into two groups of four, determined by the parity of their indices. For a fixed b_i, define $p_i(x) = b_i(a_0 + a_2\, x^{32} + a_4\, x^{64} + a_6\, x^{96})$ and $q_i(x) = b_i(a_1 + a_3\, x^{32} + a_5\, x^{64} + a_7\, x^{96})$. Here, p_i gathers the contributions of the even coefficients a_0, a_2, a_4, a_6 multiplied by b_i, while q_i gathers the odd coefficients a_1, a_3, a_5, a_7. The complete product of a by a single monomial $b_i\, x^{16i}$ is then:

$$a(x) \cdot b_i \; = p_i(x) + q_i(x) x^{16i}. \tag{1}$$

Re-indexing the full product. By summing Equation (1) over all i and expanding, the complete product can be written as a sum of the alternating p and q terms. By collecting the terms in each power x^{16k}, it is observed that position k receives contributions from both the product of the even range p_k, which was multiplied in the shift x^{16k}, and the product of the odd range q_{k-1}, which was multiplied in the shift $x^{16(k-1)}$, and from this fact carries an extra factor of x^{16} from the odd indices. From this, the expression can be assembled as follows:

$$a(x) \cdot b(x) = p_0(x) \oplus \sum_{i=1}^{7} \big(p_i(x) \oplus q_{i-1}(x)\big) x^{16i} \oplus q_7(x)\, x^{128}. \tag{2}$$

Mapping to `VMULLB` and `VMULLT`. Equation (2) maps directly to the two polynomial multiplication instructions provided by the Helium MVE extension. The `VMULLB` instruction operates on the even lanes of its source register, while `VMULLT` operates on the odd lanes, which we represent as follows: $\texttt{VMULLB}(a, b_j) = (a_0 b_j,\ a_2 b_j,\ a_4 b_j,\ a_6 b_j) = p_j(x)$, and $\texttt{VMULLT}(a, b_{j-1}) = (a_1 b_{j-1},\ a_3 b_{j-1},\ a_5 b_{j-1},\ a_7 b_{j-1}) = q_{j-1}(x)$. Performing the XOR of the two results yields exactly the coefficient pair required at step j of Equation (2): $m_j(x) = \texttt{VMULLB}(a, b_j) \oplus \texttt{VMULLT}(a, b_{j-1}) = p_j(x) \oplus q_{j-1}(x)$. Note that `VMULLT` at step j uses b_{j-1}, not b_j, because the odd coefficients already accumulated b_{j-1} in the previous step and only require b_j in the following iteration. Assigning $t_0 \leftarrow t_1$ at the end of each step ensures that $t_0 = b_{j-1}$ at the beginning of $\texttt{STEP}(j)$, allowing the value calculated in the previous iteration to be reused through a single register copy (in Algorithm 2, t_0 and t_1 correspond to r_1 and r_2, respectively).

Sliding-window accumulation. The code uses an accumulator of 16×16-bit words $r[0{:}16]$ for implementing the shifts x^{16i}, as illustrated in Figure 2. The boundary terms p_0 and q_7 are treated separately during initialization, and then a loop over $j = 1, \ldots, 7$ accumulates XOR m_j in the window $r[j{:}j{+}8]$, performing Equation (2) exactly. After all the steps, $r[0{:}8]$ contains the lower half l and $r[8{:}16]$ the upper half h, which are passed to the modular reduction. The routine has no branches, executes in constant time, and keeps a in the vector register file throughout the process; see Algorithm 2 for the implementation details and Appendix A for a description of the instructions used.

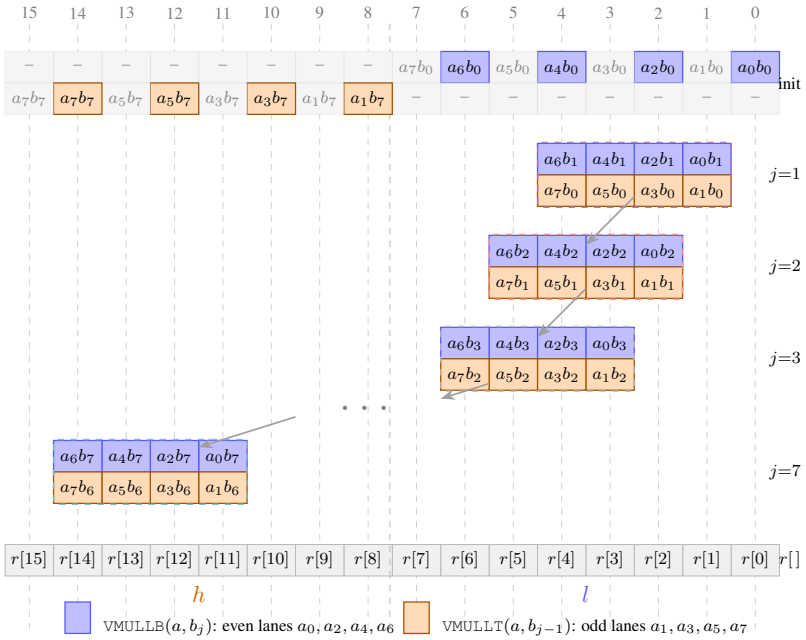

Figure 2. Sliding-window accumulation of `STEP`(j).

4.2. Hierarchical multiplication

An alternative approach implements the 128×128-bit carry-less product hierarchically, composing 32×32-bit multiplications into 64×64-bit ones, and those into the final result. At the 32-bit level, writing $a = a_H x^{16} \oplus a_L$ and $b = b_H x^{16} \oplus b_L$, the carry-less product

Algorithm 1 128-bit polynomial multiplication

Require: $a, b \in \mathrm{GF}(2^{128})$
Ensure: l, h such that $a(x) \cdot b(x) = h \cdot x^{128} \oplus l$
function MULT128X128(a, b)
  Let $a(x) = \sum_{i=0}^{7} a_i x^{16i}$
  Let $b(x) = \sum_{j=0}^{7} b_j x^{16j}$
  $c \leftarrow 0$
  for $j = 0$ **to** 7 **do**
    for $i = 0$ **to** 7 **do**
      $c \leftarrow c \oplus (a_i \cdot b_j) \cdot x^{16(i+j)}$
  $l \leftarrow c[0{:}8],\ h \leftarrow c[8{:}16]$
  return l, h

Algorithm 2 Helium MVE implementation

Require: $a, b \in \mathrm{GF}(2^{128})$
Ensure: l, h such that $a(x) \cdot b(x) = h \cdot x^{128} \oplus l$
function STEP($j, v_1[\,], v_0[\,], r_0, r_1$)
  $r_2 \leftarrow$ VDUP($v_0[j]$)
  $r_3 \leftarrow$ VMULLB(r_0, r_2)
  $r_4 \leftarrow$ VMULLT(r_0, r_1)
  $r_3 \leftarrow$ VEOR(r_3, r_4)
  $r_5 \leftarrow$ VLD1($v_1[j]$)
  $r_5 \leftarrow$ VEOR(r_5, r_3)
  VST1($v_1[j], r_5$)
  return r_2 ▷ new r_1 for next call
function MULT128X128(a, b)
  declare $v_0[8], v_1[16]$: uint16_t
  declare $r_0, \ldots, r_8$: uint32x4_t
  $r_0 \leftarrow a$
  $r_6 \leftarrow b$
  VST1(v_0, r_6)
  $r_1 \leftarrow$ VDUP($v_0[0]$)
  $r_2 \leftarrow$ VDUP($v_0[7]$)
  $r_3 \leftarrow$ VMULLB(r_0, r_1)
  $r_4 \leftarrow$ VMULLT(r_0, r_2)
  VST1($v_1[0], r_3$)
  VST1($v_1[8], r_4$)
  for $j = 1$ **to** 7 **do**
    $r_1 \leftarrow$ STEP(j, v_1, v_0, r_0, r_1)
  $r_7 \leftarrow$ VLD1($v_1[0]$)
  $r_8 \leftarrow$ VLD1($v_1[8]$)
  return $l \leftarrow r_7,\ h \leftarrow r_8$

expands as

$$a \cdot b = a_H b_H x^{32} \oplus (a_H b_L \oplus a_L b_H) x^{16} \oplus a_L b_L. \tag{3}$$

The cross term is obtained via $\bar{b} = \texttt{VREV32}(b) = b_L x^{16} \oplus b_H$, giving $t_0 = \texttt{VMULLT}(a, \bar{b}) \oplus \texttt{VMULLB}(a, \bar{b}) = a_H b_L \oplus a_L b_H$, so that the output halves are $l = \texttt{VMULLB}(a, b) \oplus (t_0 \ll 16)$ and $h = \texttt{VMULLT}(a, b) \oplus (t_0 \gg 16)$; this computation runs in parallel across all four 32-bit lanes of the vector register (Algorithm 3). At the 64-bit level, writing $a = a_H x^{32} \oplus a_L$ and $b = b_H x^{32} \oplus b_L$, the product follows the same three-term expansion $a \cdot b = a_H b_H\, x^{64} \oplus (a_H b_L \oplus a_L b_H)\, x^{32} \oplus a_L b_L$. The terms $a_L b_L$ and $a_H b_H$ are obtained from a call to MULT32X32 with (a, b), while the cross term $a_H b_L \oplus a_L b_H$ is obtained from a call to MULT32X32 with $(a, \texttt{VREV64}(b))$. The partial results are then assembled via `VEOR` and `VPSEL` with mask $p_0 =$ `0xF0F0` (Algorithm 5). Finally, the top layer applies the same decomposition to $a, b \in \mathrm{GF}(2^{128})$, with two calls to MULT64X64 producing the four 64-bit limbs r_0, r_1, r_2, r_3 assembled as $l = (r_0, r_1)$ and $h = (r_2, r_3)$ (Algorithm 6).

Binary field multiplication of 192-bit and 256-bit. The multipliers for 192-bit and 256-bit follow the same hierarchical structure as the 128-bit case, but apply an additional level of Karatsuba decomposition on the 128-bit building blocks. Both were implemented independently of the direct 128-bit Algorithm 2 and the hierarchical 128-bit Algorithm 6.

For MULT192X192, the operands $a, b \in \mathrm{GF}(2^{192})$ are divided into $a = a_H x^{128} \oplus a_L$ and $b = b_H x^{128} \oplus b_L$, where $a_L, b_L \in \mathrm{GF}(2^{128})$ and $a_H, b_H \in \mathrm{GF}(2^{64})$. The three Karatsuba subproducts $p_0 = a_L b_L$, $p_2 = a_H b_H$, and $p_1 = (a_L \oplus a_H)(b_L \oplus b_H)$ are computed via MULT128X128 and MULT64X64, respectively, and combined to produce the 384-bit result: $a \cdot b = p_2\, x^{256} \oplus (p_2 \oplus p_0 \oplus p_1)\, x^{128} \oplus p_0$.

For MULT256X256, both operands belong to $\mathrm{GF}(2^{256})$ and are similarly divided as $a = a_H x^{128} \oplus a_L$ and $b = b_H x^{128} \oplus b_L$, where $a_L, b_L, a_H, b_H \in \mathrm{GF}(2^{128})$. The carry-less product expands as $a \cdot b = a_H b_H\, x^{256} \oplus (a_H b_L \oplus a_L b_H)\, x^{128} \oplus a_L b_L = p_2\, x^{256} \oplus (p_2 \oplus p_0 \oplus p_1)\, x^{128} \oplus p_0$, where $p_0 = a_L b_L$, $p_2 = a_H b_H$, and $p_1 = (a_H \oplus a_L)(b_H \oplus b_L)$ are the three Karatsuba subproducts, all evaluated by MULT128X128. Setting $m = p_1 \oplus p_0 \oplus p_2$, the 512-bit result (c_0, c_1, c_2, c_3) is assembled as $c_0 = p_0^{lo}$, $c_1 = p_0^{hi} \oplus m^{lo}$, $c_2 = p_2^{lo} \oplus m^{hi}$,

Algorithm 3 Four parallel MULT32X32 multiplications

Require: a, b: 128-bit registers viewed as four 32-bit lanes
Ensure: l: lower 32-bit half of each lane product, h: upper 32-bit half of each lane product
1: **function** MULT32X32(a, b)
2: $r_0 \leftarrow$ VREV32(b)
3: $r_1 \leftarrow$ VMULLT(a, r_0)
4: $r_2 \leftarrow$ VMULLB(a, r_0)
5: $r_3 \leftarrow$ VEOR(r_1, r_2)
6: $r_4 \leftarrow$ VMULLT(a, b)
7: $r_5 \leftarrow$ VMULLB(a, b)
8: $r_6 \leftarrow$ VSHL(r_3, 16)
9: $r_7 \leftarrow$ VSHR(r_3, 16)
10: $l \leftarrow$ VEOR(r_5, r_6)
11: $h \leftarrow$ VEOR(r_4, r_7)
12: **return** l , h

Algorithm 4 One MULT64X64

Require: a, b: 128-bit registers viewed as two 64-bit lanes
Ensure: l: lower 64-bit half of the product, h: upper 64-bit half of the product
1: **function** MULT64X64_1(a, b)
2: $r_0 \leftarrow a$
3: VST1(v_0, b)
4: $r_1 \leftarrow$ VDUP($v_0[0]$)
5: $r_2 \leftarrow$ VDUP($v_0[3]$)
6: $r_3 \leftarrow$ VMULLB(r_0, r_1)
7: $r_4 \leftarrow$ VMULLT(r_0, r_2)
8: VST1($v_1[0]$, r_3)
9: VST1($v_1[4]$, r_4)
10: STEP(1); STEP(2); STEP(3) ▷ same as Algorithm 2
11: $r_5 \leftarrow$ VLD1($v_1[0]$)
12: $r_6 \leftarrow$ VLD1($v_1[4]$)
13: **return** $l \leftarrow r_5$, $h \leftarrow r_6$

Algorithm 5 Two parallel MULT64X64 multiplications

Require: a, b: 128-bit registers viewed as two 64-bit lanes
Ensure: l: lower halves of both lane products, h: upper halves of both lane products
1: **function** MULT64X64(a, b)
2: $r_0 \leftarrow a$
3: $r_1 \leftarrow b$
4: $r_2 \leftarrow$ VREV64(r_1)
5: MULT32x32(r_0, r_2, &r_3, &r_4)
6: MULT32x32(r_0, r_1, &r_5, &r_6)
7: $r_7 \leftarrow$ VEOR(r_6, r_3)
8: $r_8 \leftarrow$ VEOR(r_5, r_4)
9: $r_7 \leftarrow$ VREV64(r_7)
10: $r_8 \leftarrow$ VREV64(r_8)
11: $r_7 \leftarrow$ VEOR(r_7, r_3)
12: $r_8 \leftarrow$ VEOR(r_8, r_4)
13: $r_9 \leftarrow$ VPSEL(r_5, r_7, p_0)
14: $r_{10} \leftarrow$ VPSEL(r_8, r_6, p_0)
15: **return** $l \leftarrow r_{10}$, $h \leftarrow r_9$

Algorithm 6 128-bit hierarchical multiplication

Require: $a, b \in \mathrm{GF}(2^{128})$
Ensure: l, h such that $a(x) \cdot b(x) = h \cdot x^{128} \oplus l$
1: **function** MULT128X128_HIER(a, b)
2: $r_0 \leftarrow$ VGET(b, 0)
3: $r_1 \leftarrow$ VGET(b, 1)
4: $tmp \leftarrow (r_0, r_1)$
5: MULT64x64(a, tmp, &r_2, &r_3)
6: $tmp \leftarrow (r_1, r_0)$
7: MULT64x64(a, tmp, &r_4, &r_5)
8: $r_6 \leftarrow$ VGET(r_3, 0)
9: $r_7 \leftarrow$ VGET(r_2, 1)
10: $r_8 \leftarrow$ VGET(r_2, 0) $\oplus$ VGET(r_5, 0) $\oplus$ VGET(r_5, 1)
11: $r_9 \leftarrow$ VGET(r_3, 1) $\oplus$ VGET(r_4, 0) $\oplus$ VGET(r_4, 1)
12: $l \leftarrow (r_6, r_8)$
13: $h \leftarrow (r_9, r_7)$
14: **return** l , h

and $c_3 = p_2^{hi}$, where the superscripts lo and hi denote the lower and upper 128-bit halves of each 256-bit subproduct.

4.3. Squaring

Squaring in $\mathrm{GF}(2^m)$ is a specialization of multiplication that eliminates all cross-products. For $a(x) = \sum_{i=0}^{7} a_i\, x^{16i}$ with $a_i \in \mathrm{GF}(2^{16})$, the Frobenius endomorphism gives $a(x)^2 = \sum_{i=0}^{7} a_i^2\, x^{32i}$, so each 16-bit coefficient is squared independently and placed at position x^{32i}, with no interaction between coefficients. Splitting into lower and upper halves:

$$a(x)^2 = \sum_{i=0}^{3} a_i^2\, x^{32i} \oplus \left(\sum_{i=0}^{3} a_{4+i}^2\, x^{32i} \right) x^{128}. \tag{4}$$

To map this computation to `VMULLB` and `VMULLT`, the vector $a = (a_0, a_1, \ldots, a_7)$ is first permuted to $(a_0, a_4, a_1, a_5, a_2, a_6, a_3, a_7)$ using `VREV64` and `VPSEL`. In the permuted vector, the even lanes (a_0, a_1, a_2, a_3) and odd lanes (a_4, a_5, a_6, a_7) are separated, so that `VMULLB` and `VMULLT` with the permuted vector as both operands yield $p_{sq} = (a_0^2, a_1^2, a_2^2, a_3^2)$ and $q_{sq} = (a_4^2, a_5^2, a_6^2, a_7^2)$, corresponding to the lower and upper halves of Equation (4), respectively.

For $\mathrm{GF}(2^{192})$ and $\mathrm{GF}(2^{256})$, the same procedure is applied independently to each 128-bit block of the input. Since each a_i^2 occupies exactly the 32 bits at position x^{32i}, squared coefficients land in disjoint bit positions and no overlap between blocks needs to be combined. For $\mathrm{GF}(2^{192})$, the high block contains only 64 bits of data, so lane 2 is zeroed before the rearrangement. The partial results are then passed to the respective reduction routine; see Algorithms 8, 9, and 10.

Algorithm 7 128-bit polynomial squaring

Require: $a \in \mathrm{GF}(2^{128})$
Ensure: l, h such that $a(x)^2 = h \cdot x^{128} \oplus l$
function SQ128(a)
  Write $a(x) = \sum_{i=0}^{7} a_i\, x^{16i}$
  $c \leftarrow 0$
  for $i = 0$ **to** 7 **do**
    $c \leftarrow c \oplus a_i^2 \cdot x^{32i}$
  return $l \leftarrow c[0{:}8]$, $h \leftarrow c[8{:}16]$

Algorithm 8 Squaring in $\mathrm{GF}(2^{128})$

Require: $a \in \mathrm{GF}(2^{128})$
Ensure: $c = a^2 \in \mathrm{GF}(2^{128})$
function GF128_SQUARE(a)
  $r_0 \leftarrow$ VGET(a, 1)
  $r_1 \leftarrow$ VGET(a, 2)
  $r_2 \leftarrow$ VSET(r_0, a, 2)
  $r_2 \leftarrow$ VSET(r_1, r_2, 1)
  $r_3 \leftarrow$ VREV64(r_2)
  $r_4 \leftarrow$ VPSEL(r_2, r_3, 0xC3C3)
  $r_5 \leftarrow$ VMULLB(r_4, r_4)
  $r_6 \leftarrow$ VMULLT(r_4, r_4)
  GF128_REDUCTION(r_6, r_5, &c)

Algorithm 9 Squaring in $\mathrm{GF}(2^{192})$

Require: h, l: high and low 128-bit halves of $a \in \mathrm{GF}(2^{192})$
Ensure: c_0, c_1: low and high halves of a^2
function GF192_SQUARE(h, l)
  $r_0 \leftarrow$ VGET(l, 1)
  $r_1 \leftarrow$ VGET(l, 2)
  $r_2 \leftarrow$ VSET(r_0, l, 2)
  $r_2 \leftarrow$ VSET(r_1, r_2, 1)
  $r_3 \leftarrow$ VREV64(r_2)
  $r_4 \leftarrow$ VPSEL(r_2, r_3, 0xC3C3)
  $r_5 \leftarrow$ VMULLB(r_4, r_4)
  $r_6 \leftarrow$ VMULLT(r_4, r_4)
  $r_7 \leftarrow$ VGET(h, 1)
  $r_8 \leftarrow$ VSET(r_7, h, 2)
  $r_8 \leftarrow$ VSET(0, r_8, 1)
  $r_9 \leftarrow$ VREV64(r_8)
  $r_{10} \leftarrow$ VPSEL(r_8, r_9, 0xC3C3)
  $r_{11} \leftarrow$ VMULLB(r_{10}, r_{10})
  GF192_REDUCTION(r_{11}, r_6, r_5, &c_1, &c_0)

Algorithm 10 Squaring in $\mathrm{GF}(2^{256})$

Require: h, l: high and low 128-bit halves of $a \in \mathrm{GF}(2^{256})$
Ensure: c_0, c_1: low and high halves of a^2
function GF256_SQUARE(h, l)
  $r_0 \leftarrow$ VGET(l, 1)
  $r_1 \leftarrow$ VGET(l, 2)
  $r_2 \leftarrow$ VSET(r_0, l, 2)
  $r_2 \leftarrow$ VSET(r_1, r_2, 1)
  $r_3 \leftarrow$ VREV64(r_2)
  $r_4 \leftarrow$ VPSEL(r_2, r_3, 0xC3C3)
  $r_5 \leftarrow$ VMULLB(r_4, r_4)
  $r_6 \leftarrow$ VMULLT(r_4, r_4)
  $r_7 \leftarrow$ VGET(h, 1)
  $r_8 \leftarrow$ VGET(h, 2)
  $r_9 \leftarrow$ VSET(r_7, h, 2)
  $r_9 \leftarrow$ VSET(r_8, r_9, 1)
  $r_{10} \leftarrow$ VREV64(r_9)
  $r_{11} \leftarrow$ VPSEL(r_9, r_{10}, 0xC3C3)
  $r_{12} \leftarrow$ VMULLB(r_{11}, r_{11})
  $r_{13} \leftarrow$ VMULLT(r_{11}, r_{11})
  GF256_REDUCTION(r_{13}, r_{12}, r_6, r_5, &c_1, &c_0)

4.4. Modular reduction

After a carry-less multiplication or squaring, the unreduced result has degree up to $2m - 2$ and must be reduced modulo the irreducible polynomial $f(x)$ to obtain an element of $\mathrm{GF}(2^m)$. Let the unreduced product be written as $c(x) = c_L(x) + x^m c_H(x)$, where $c_L, c_H \in \mathrm{GF}(2^m)$ denote the lower and higher halves, respectively. Using the defining relation $f(x) = x^m + r(x)$, we substitute $x^m \equiv r(x)$ and obtain $c(x) \bmod f(x) = c_L(x) \oplus c_H(x)\, r(x)$.

Since $r(x)$ has low degree, the product $c_H(x)\, r(x)$ may still produce terms of degree $\geq m$, requiring a second reduction step. The reduction therefore follows a two-stage structure: $c(x) \bmod f(x) = c_L(x) \oplus l(x) \oplus h(x)$, where $l(x)$ contains the lower terms

of $c_H(x)\, r(x)$ and $h(x)$ the overflow terms that must be reduced once more. Interpreting c_H as eight 16-bit coefficients, the product $c_H(x)\, r(x)$ is computed using the same even/odd lane decomposition as in the multiplication step: $l = \texttt{VMULLB}(c_H, \texttt{mod})$ and $h = \texttt{VMULLT}(c_H, \texttt{mod})$, where $\texttt{mod}$ encodes the polynomial $r(x)$.

The high part h corresponds to terms multiplied by x^{16} and must be shifted right by one 16-bit lane before accumulation. This shift is implemented without an explicit instruction: h is stored in an aligned buffer at offset 1 via $\texttt{vst1}(\&v[1], h)$ and reloaded from offset 0 via $\texttt{vld1}(\&v[0])$, which places a zero in lane 0 and shifts the remaining lanes by one position. The shifted value is XOR-ed into l, and the residual high word at $v[8]$ is reduced by a second multiplication by $\texttt{mod}$ and XOR-ed in. The final reduced result is $c(x) \bmod f(x) = c_L(x) \oplus \texttt{VMULLB}(c_H, \texttt{mod}) \oplus \texttt{shift}\big(\texttt{VMULLT}(c_H, \texttt{mod})\big) \oplus \texttt{VMULLB}(v[8], \texttt{mod})$. For $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, the constants $\texttt{mod} = \texttt{0x0087}$, $\texttt{0x0087}$, and $\texttt{0x0425}$ are used, respectively; the corresponding algorithms are given in Algorithms 12, 13, and 14.

We also evaluated a mixed implementation that combines Helium vector instructions with scalar Arm instructions for parts of the reduction, motivated by the fact that the Cortex-M85 maintains separate pipelines for vector and scalar operations, which in principle allows overlap between the two. However, in practice, the fully vectorized implementation outperformed the mixed variant across all field sizes, as the overhead of moving data between the scalar and vector register files and the synchronization cost between the two pipelines exceeded any benefit from the attempted interleaving. As a result, all reductions in the final implementation use only Helium vector instructions.

5. Application to post-quantum cryptography

The field arithmetic primitives described in the previous section are evaluated in the context of two post-quantum signature schemes that rely heavily on binary field multiplication: FAEST and AIMer. Both schemes require arithmetic over $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, and are described below.

5.1. FAEST

FAEST [Baum et al. 2025] is a post-quantum digital signature scheme submitted to the NIST additional signature standardization process [NIST 2022]. Its security relies solely on the one-wayness of AES: the secret signing key is an AES key sk, and the public verification key is a pair of plaintext-ciphertext (x, y) such that $E_{sk}(x) = y$. A signature is a non-interactive zero-knowledge argument of knowledge of sk, constructed by combining the VOLE-in-the-Head (VOLEitH) paradigm with the QuickSilver proof system [Baum et al. 2023], made non-interactive via the Fiat–Shamir transform. The dominant computational cost of FAEST arises from evaluating the AES circuit inside the QuickSilver proof: both the S-box inversion and the MixColumns step require multiplications in $\mathrm{GF}(2^8)$, which are then lifted to the extension field $\mathrm{GF}(2^\lambda)$ used by the VOLE correlations so that the proof system can verify them. Key and signature sizes for all parameter sets are listed in Table 1.

5.2. AIMer

AIMer [Ha et al. 2024] is a post-quantum digital signature scheme standardized by the Korean post-quantum competition KpqC in January 2025 [Firmin 2025]. Its security is

Algorithm 11 Modular reduction

Require: $c(x) = c_L(x) \oplus x^{128}\, c_H(x)$, irreducible $f(x) = x^{128} + r(x)$
Ensure: $c(x) \bmod f(x)$
function REDUCTION($c(x)$, $f(x)$)
  $t(x) \leftarrow c_H(x) \cdot r(x)$
  $l(x) \leftarrow t(x) \bmod x^{128}$
  $h(x) \leftarrow \lfloor t(x)/x^{128} \rfloor$
  if $c_H(x) \neq 0$ **then**
    $l(x) \leftarrow l(x) \oplus h(x) \cdot r(x)$
  return $c_L(x) \oplus l(x)$

Algorithm 12 Reduction in $\mathrm{GF}(2^{128})$

Require: $high, low$: upper and lower halves of unreduced product
Ensure: $red = (high \cdot x^{128} \oplus low) \bmod f(x)$
function GF128_REDUCTION($high$, low)
  $r_0 \leftarrow$ VDUP(0x0087)
  $r_1 \leftarrow$ VMULLB($high$, r_0)
  $r_2 \leftarrow$ VMULLT($high$, r_0)
  VST1($v_0[1]$, r_2)
  $r_3 \leftarrow$ VLD1($v_0[0]$)
  $r_1 \leftarrow$ VEOR(r_1, r_3)
  $r_4 \leftarrow$ VLD1($v_0[8]$)
  $r_5 \leftarrow$ VMULLB(r_4, r_0)
  $r_1 \leftarrow$ VEOR(r_1, r_5)
  return VEOR(r_1, low)

Algorithm 13 Reduction in $\mathrm{GF}(2^{192})$

Require: $high_0, low_1, low_0$: limbs of unreduced product
Ensure: h_0, h_1: reduced result in $\mathrm{GF}(2^{192})$
function GF192_REDUCTION($high_0$, low_1, low_0)
  $r_0 \leftarrow$ VDUP(0x0087)
  $r_1 \leftarrow$ VCREATE(lane$_1$(low_1), lane$_0$($high_0$))
  $r_2 \leftarrow$ VCREATE(lane$_1$($high_0$), 0)
  $r_3 \leftarrow$ VMULLB(r_2, r_0)
  $r_4 \leftarrow$ VMULLT(r_2, r_0)
  VST1($v_0[1]$, r_4)
  $r_5 \leftarrow$ VLD1($v_0[0]$)
  $r_3 \leftarrow$ VEOR(r_3, r_5)
  $r_6 \leftarrow$ VLD1($v_0[4]$)
  $r_1 \leftarrow$ VEOR(r_1, r_6)
  $r_7 \leftarrow$ VMULLB(r_1, r_0)
  $r_8 \leftarrow$ VMULLT(r_1, r_0)
  VST1($v_0[1]$, r_8)
  $r_9 \leftarrow$ VLD1($v_0[0]$)
  $r_7 \leftarrow$ VEOR(r_7, r_9)
  $r_{10} \leftarrow$ VLD1($v_0[8]$)
  $r_3 \leftarrow$ VEOR(r_3, r_{10})
  $r_{11} \leftarrow$ VEOR(r_7, low_0)
  $r_{12} \leftarrow$ VEOR(r_3, low_1)
  $r_{12} \leftarrow$ VSET(0, r_{12}, 2)
  $r_{13} \leftarrow$ VSET(0, r_{12}, 3)
  return $h_0 \leftarrow r_{11},\ h_1 \leftarrow r_{13}$

Algorithm 14 Reduction in $\mathrm{GF}(2^{256})$

Require: $high_1, high_0, low_1, low_0$: limbs of unreduced product
Ensure: h_0, h_1: reduced result in $\mathrm{GF}(2^{256})$
function GF256_REDUCTION($high_1$, $high_0$, low_1, low_0)
  $r_0 \leftarrow$ VDUP(0x0425)
  $r_1 \leftarrow$ VMULLB($high_1$, r_0)
  $r_2 \leftarrow$ VMULLT($high_1$, r_0)
  VST1($v_0[1]$, r_2)
  $r_3 \leftarrow$ VLD1($v_0[0]$)
  $r_1 \leftarrow$ VEOR(r_1, r_3)
  $r_4 \leftarrow$ VLD1($v_0[8]$)
  $r_5 \leftarrow$ VEOR($high_0$, r_4)
  $r_6 \leftarrow$ VMULLB(r_5, r_0)
  $r_7 \leftarrow$ VMULLT(r_5, r_0)
  VST1($v_0[1]$, r_7)
  $r_8 \leftarrow$ VLD1($v_0[0]$)
  $r_6 \leftarrow$ VEOR(r_6, r_8)
  $r_9 \leftarrow$ VLD1($v_0[8]$)
  $r_1 \leftarrow$ VEOR(r_1, r_9)
  $r_{10} \leftarrow$ VEOR(r_6, low_0)
  $r_{11} \leftarrow$ VEOR(r_1, low_1)
  return $h_0 \leftarrow r_{10},\ h_1 \leftarrow r_{11}$

based on the one-wayness of AIM2 [Kim et al. 2023], a symmetric primitive designed to be efficient in the MPC-in-the-Head (MPCitH) [Ishai et al. 2007] setting. AIM2 takes a secret plaintext $pt \in \mathrm{GF}(2^n)$ and a public initialization vector iv, and produces a public ciphertext ct through $\ell \in \{2, 3\}$ parallel branches, each consisting of an inverse Mersenne S-box of the form $x \mapsto x^{\bar{e}}$, where $\bar{e} = (2^e - 1)^{-1} \bmod 2^n - 1$, followed by an affine layer and a final Mersenne output S-box. The number of branches ℓ depends on the security level: $\ell = 2$ for $n \in \{128, 192\}$ and $\ell = 3$ for $n = 256$. The inverse Mersenne S-box is the only nonlinear component of AIM2, and each evaluation decomposes into a chain of squarings and a single field multiplication over $\mathrm{GF}(2^n)$, making binary field arithmetic the dominant cost of every AIM2 call. AIMer is defined at three security levels for $n \in \{128, 192, 256\}$, requiring arithmetic over $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, respectively, with these operations invoked repeatedly during key generation, signing, and verification. Key and signature sizes for all parameter sets are listed in Table 1.

Table 1. Key and signature sizes (B) for AIMer, FAEST, and FAEST-EM.

	Security Level 1 (128-bit)			Security Level 3 (192-bit)			Security Level 5 (256-bit)		
Variant	pk	sk	sig	pk	sk	sig	pk	sk	sig
AIMer-f	32	48	5,888	48	72	13,056	64	96	25,120
AIMer-s	32	48	4,160	48	72	9,120	64	96	17,056
FAEST-f	32	32	5,924	40	48	14,948	48	48	26,548
FAEST-s	32	32	4,506	40	48	11,260	48	48	20,696
FAEST-EM-f	32	32	5,060	48	48	12,380	64	64	23,476
FAEST-EM-s	32	32	3,906	48	48	9,340	64	64	17,984

6. Results

Our AIMer and FAEST implementations are based on Cortex-M4 optimized implementations by [Kwon et al. 2025] and by [Aranha et al. 2025], distributed via the `pqm4` benchmarking framework [Kannwischer et al. 2019] and the `pqm4-faest` repository, respectively. Both originally target the Cortex-M4. The adaptation for the Cortex-M85 required two main changes: realigning the data structures to 16-byte boundaries to meet the alignment requirements of the MVE load and store instructions, and converting all intermediate buffers from dynamically allocated arrays to statically allocated arrays. The latter was necessary because the Renesas EK-RA8M1 platform requires data to reside in known memory regions for cache-coherent access; dynamic allocation via malloc places buffers in regions that may fall outside the fast SRAM window, introducing unpredictable latency penalties during field arithmetic. Aside from these changes, no modifications to the algorithmic structure of either scheme were necessary.

Regarding FAEST, the reference implementation for 32-bit targets version 1 of the scheme. To obtain results for version 2, we applied the updated parameter sets and domain separation constants from the v2 [Baum et al. 2025] specification directly to the adapted source code. No new code was written for this purpose: the transition was based exclusively on the existing 32-bit base algorithm and the reference implementation, combining them to produce the v2 results. The underlying proof system and field arithmetic structure remain unchanged between versions.

Cycle counts were collected on the Renesas EK-RA8M1 board at its maximum frequency of 480 MHz. The device features 1 MB of internal SRAM, divided into 64 KB ITCM, 64 KB DTCM, and 896 KB main SRAM. To reduce instruction fetch latency, the field multiplication and squaring routines, as well as other performance-critical functions, were placed in the Instruction Tightly Coupled Memory (ITCM) via linker directives, while the remaining code executes from internal flash. The execution stack is allocated in main SRAM with data caching enabled. Two compiler chains were evaluated: arm-none-eabi-gcc 13.3.1 and Clang/LLVM 19.1.5, both targeting arm-none-eabi with optimization level -O3. Measurements at -O2 produced nearly identical cycle counts and are therefore omitted. This behavior is expected because the proposed field arithmetic routines contain no secret-dependent branches or memory accesses, leaving the compiler with no data-dependent control flow to alter regardless of the optimization level. Disassembly of the 128-bit multiplication and reduction kernels compiled at `-O2` and `-O3` produced identical generated code in both cases, confirming that the branch-free structure, and therefore the constant-time property, is preserved independently of the optimization level. Benchmarks

were carried out following the full-cache methodology adopted by [Aranha et al. 2025]: the structure responsible for managing VOLE correlations operates in full-cache mode, with the complete matrix computed and kept in memory in a single pass and no partial recomputation, both during signing and verification, making the results directly comparable to those reported in the reference literature. To avoid measurement bias from cache or branch-predictor effects, a new message was generated on every iteration prior to measuring signing and verification. The arithmetic microbenchmarks report the median cycle count over 10 000 iterations for AIMer and 100 iterations for FAEST.

6.1. Field Arithmetic

Table 2 shows the cycle count for the individual field arithmetic primitives on the Cortex-M85. For modular reduction the fully vectorized implementation achieves 9–11 cycles for $\mathrm{GF}(2^{128})$, 32–34 cycles for $\mathrm{GF}(2^{192})$, and 27–36 cycles for $\mathrm{GF}(2^{256})$. As discussed in Section 4.4, a mixed vector–scalar variant of the reduction was also evaluated but did not outperform the fully vectorized implementation on any field size.

For squaring, the proposed approach in Section 4.3 reduces the operation to a register rearrangement followed by two carry-less multiplication instructions, achieving 10–12 cycles for $\mathrm{GF}(2^{128})$, 14–15 cycles for $\mathrm{GF}(2^{192})$, and 24–25 cycles for $\mathrm{GF}(2^{256})$ without reduction. Including reduction, the total cost reaches 23–25 cycles for $\mathrm{GF}(2^{128})$, 51–53 cycles for $\mathrm{GF}(2^{192})$, and 45–46 cycles for $\mathrm{GF}(2^{256})$. Compared to the AIMer scalar reference implementation, the optimized squaring is up to 3.30× faster for $\mathrm{GF}(2^{128})$, 2.08× faster for $\mathrm{GF}(2^{192})$, and 3.13× faster for $\mathrm{GF}(2^{256})$.

For multiplication, two approaches were implemented and compared: the direct algorithm (Algorithm 2) and the hierarchical algorithm (Algorithm 6). For $\mathrm{GF}(2^{128})$, the direct algorithm achieves 77–110 cycles including reduction, while the hierarchical algorithm requires 98–123 cycles. The direct algorithm is consistently faster across all field sizes, achieving speedups of up to 1.36× for $\mathrm{GF}(2^{192})$ and 1.53× for $\mathrm{GF}(2^{256})$ over the hierarchical approach.

These results confirm that the direct algorithm is the best choice for all three field sizes and, therefore, has been adopted in the implementations of the signature schemes. Compared to scalar reference implementations, the Helium-optimized direct multiplication is up to 10.8× faster than FAEST [Aranha et al. 2025] and up to 8.4× faster than AIMer [Kwon et al. 2025] for $\mathrm{GF}(2^{128})$, with similar gains at the 192-bit and 256-bit security levels.

6.2. Signature Scheme Performance

Table 3 shows the performance of AIMer at the three security levels and in both parameter variants (fast and short), and Table 4 shows the performance of FAEST and FAEST-EM at the same security levels. In both Tables 3 and 4, the “Reference” column corresponds to the best result obtained by the original implementation targeting M85 in both compilers, and the “This work” column corresponds to the best result of the optimized implementation for Helium, to date.

AIMer. The Helium implementation achieves speedups in all operations and security levels, as shown in Table 3. Key generation is up to 1.33× faster. For the fast variant,

Table 2. Cycle counts for field arithmetic on the Cortex-M85 (`-O3`).

Modular Reduction (This Work)

Operation	GCC	Clang
$Reduction(128)$ (Alg. 12)	11	9
$Reduction(192)$ (Alg. 13)	34	32
$Reduction(256)$ (Alg. 14)	36	27

Field Squaring (w/o reduction, This Work)

Operation	GCC	Clang
128^2 (Alg. 8)	10	12
192^2 (Alg. 9)	14	15
256^2 (Alg. 10)	24	25

Field Multiplication (w/o reduction, This Work)

Operation	GCC	Clang
128×128 direct (Alg. 2)	102	73
$4\times$ Mult 32×32 (Alg. 3)	17	14
$2\times$ Mult 64×64 (Alg. 5)	40	40
$1\times$ Mult 64×64	44	33
128×128 hierarchical (Alg. 6)	108	88
192×192 direct	348	169
192×192 hierarchical	374	254
256×256 direct	376	195
256×256 hierarchical	511	279

Field Squaring & Multiplication (with reduction)

	$\mathrm{GF}(2^{128})$		$\mathrm{GF}(2^{192})$		$\mathrm{GF}(2^{256})$	
Implementation	GCC	Clang	GCC	Clang	GCC	Clang
Squaring						
This Work (Helium)	23	25	53	51	45	46
AIMer [Kwon et al. 2025]	73	76	103	106	139	141
Multiplication						
This Work – direct	110	77	371	219	475	228
This Work – hierarchical	123	98	410	298	636	349
FAEST [Aranha et al. 2025]	915	832	1611	1573	2970	2130
AIMer [Kwon et al. 2025]	676	643	1390	1299	2157	2041

signing and verification reach speedups of up to $1.35\times$ and $1.34\times$, respectively. For the short variant, speedups range from $1.14\times$ to $1.35\times$. The gains are most pronounced for the 256-bit security level, where field multiplications over $\mathrm{GF}(2^{256})$ represent a larger fraction of the total execution time, and the advantage of Helium instructions is most evident.

Table 3. AIMer performance on Cortex-M85 (`-O3`, cycles$/10^6$).

		Reference			This work			Speedup		
Var.	Op.	128	192	256	128	192	256	128	192	256
–	Keygen	0.341	0.817	2.004	0.257	0.652	1.665	1.33×	1.25×	1.20×
F	Sign	18.821	49.095	104.234	15.904	40.465	77.164	1.18×	1.21×	1.35×
	Verify	17.762	45.919	97.979	14.900	38.382	73.099	1.19×	1.20×	1.34×
S	Sign	185.761	483.815	900.029	163.009	415.268	694.007	1.14×	1.17×	1.30×
	Verify	148.071	388.831	773.040	124.971	314.591	573.884	1.18×	1.24×	1.35×

FAEST. The results for FAEST are shown in Table 4. For standard FAEST, speedups range from $1.10\times$ to $1.23\times$ for the fast variant and from $1.01\times$ to $1.05\times$ for the short variant. The gains are more modest than those observed for AIMer because AIM2 is built directly over $\mathrm{GF}(2^{\lambda})$, while FAEST uses AES (which operates in $\mathrm{GF}(2^8)$) as its one-way function and only relies on $\mathrm{GF}(2^{\lambda})$ arithmetic in the surrounding zero-knowledge proof. For FAEST-EM, which reduces the number of AES evaluations and therefore increases the relative contribution of field arithmetic to the total cost, the gains are more significant, reaching $1.30\times$ for 256-bit level signatures in the fast variant. FAEST key generation data was omitted, as the specific $\mathrm{GF}(2^8)$ finite field used for signature generation was not the focus of this optimization work.

Finally, Clang consistently produces better code than GCC for Helium-specific

Table 4. FAEST and FAEST-EM performance on Cortex-M85 (`-O3`, cycles$/10^6$).

		Reference			This work			Speedup		
Var.	Op.	128	192	256	128	192	256	128	192	256
FAEST										
F	Sign	116.067	347.990	905.136	98.472	282.519	785.217	1.18×	1.23×	1.15×
	Verify	101.285	323.910	849.887	89.046	279.431	773.228	1.14×	1.16×	1.10×
S	Sign	795.240	2754.356	6746.262	759.076	2693.358	6642.785	1.05×	1.02×	1.02×
	Verify	704.277	2234.950	6519.935	691.651	2196.711	6451.861	1.02×	1.02×	1.01×
FAEST-EM										
F	Sign	67.589	205.554	469.370	52.006	166.717	361.088	1.30×	1.23×	1.30×
	Verify	54.844	189.556	388.928	44.557	156.794	317.186	1.23×	1.21×	1.23×
S	Sign	484.844	1556.484	3401.486	469.394	1511.877	3291.342	1.03×	1.03×	1.03×
	Verify	366.942	1471.392	2774.508	356.771	1441.622	2699.503	1.03×	1.02×	1.03×

intrinsic functions in all benchmarks, in some cases reducing the cycle count by more than 15% for the same source code. This suggests that the instruction scheduling quality for MVE instructions varies significantly between compiler toolchains and that Clang should be preferred for Helium-targeted implementations. Although individual field arithmetic primitives show significant reductions in cycle count compared to scalar reference implementations, these do not directly translate to the overall signature scheme performance. When integrated with AIMer and FAEST, the overall speed gains are more modest, as the algorithms as a whole have other dependencies, which make the gains moderate.

7. Conclusion

In this work, we present fully vectorized and constant-time implementations of algorithms for multiplication and squaring in $\mathrm{GF}(2^{128})$, $\mathrm{GF}(2^{192})$, and $\mathrm{GF}(2^{256})$, targeting the Helium instruction set on the Arm Cortex-M85 processor. The proposed multiplication algorithm exploits the even/odd lane decomposition of the VMULLB and VMULLT instructions through a sliding-window accumulation scheme, and has proven to be consistently faster than a hierarchical approach based on 32×32-bit and 64×64-bit compositions in all three field sizes.

To evaluate the practical impact of these primitives, we integrated them into the AIMer and FAEST post-quantum signature schemes. For AIMer, the Helium-optimized implementation is up to 1.35× faster than the M4-targeted reference, with the most significant gains observed at the 256-bit security level, where field arithmetic represents a larger fraction of the total cost. For FAEST, the gains are more modest, reaching up to 1.23× for the standard variant and 1.30× for FAEST-EM. These results demonstrate that the Helium extension of the Cortex-M85 is a suitable platform for efficient and secure binary field arithmetic in embedded devices, and that it enables competitive performance for post-quantum signature schemes whose cost is dominated by field operations.

Future Work. As future work, we plan to extend the Helium optimization to the remaining components of the FAEST implementation, including the SHAKE-based hash functions, with the goal of developing a fully Helium-native implementation of the FAEST signature scheme and further improving its performance, security, and efficiency.

Acknowledgments

The first, third, and fourth authors gratefully acknowledge the Technology Innovation Institute for its partial financial support during the development of this work. The authors also thank the anonymous reviewers for their valuable suggestions and constructive feedback.

References

Alagic, G., Bros, M., Ciadoux, P., Cooper, D., Dang, Q., Dang, T., Kelsey, J. M., Lichtinger, J., Miller, C. A., Moody, D., Peralta, R., Perlner, R., Robinson, A., Silberg, H., Smith-Tone, D., Waller, N., and Liu, Y.-K. (2024). Status report on the first round of the additional digital signature schemes for the NIST post-quantum cryptography standardization process. Technical Report NIST IR 8528, National Institute of Standards and Technology (NIST), Gaithersburg, MD.

Aranha, D. F., Degn, J., Eilath, J., Nielsen, K., and Scholl, P. (2025). FAEST for memory-constrained devices with side-channel protections. Cryptology ePrint Archive, Paper 2025/1261.

Arm Ltd. (2023). *Armv8-M Architecture Reference Manual – Mainline and Helium Extension*. `https://developer.arm.com/documentation/ddi0553/latest/armv8-m-architecture-reference-manual`. Accessed: 2025-08-02.

Baum, C., Beullens, W., Braun, L., de Saint Guilhem, C. D., Klooß, M., Majenz, C., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., and Scholl, P. (2025). Faest v2: Algorithm specifications. Version 2.0. National Institute of Standards and Technology (NIST). `https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/faest-spec-round2-web.pdf`.

Baum, C., Braun, L., de Saint Guilhem, C. D., Klooß, M., Orsini, E., Roy, L., and Scholl, P. (2023). Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In Handschuh, H. and Lysyanskaya, A., editors, *Advances in Cryptology – CRYPTO 2023*, pages 581–615, Cham. Springer Nature Switzerland.

Becker, H., Hwang, V., Kannwischer, M., Yang, B.-Y., and Yang, S.-Y. (2021). Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 221–244.

Firmin, C. (2025). South Korea announces winners of KpqC competition. PQShield. `https://pqshield.com/south-korea-announces-winners-of-kpqc-competition/`.

Ha, J., Kim, S., Kwon, J., Lee, B., Lee, J., Lee, J., Lee, S., Moon, D., Son, M., Cho, J., and Yoon, H. (2024). The AIMer signature scheme, version 2.1. Technical report, Submission to the KpqC Competition. `https://aimer-signature.org/docs/AIMer-specification-v2.1.pdf`.

Ishai, Y., Kushilevitz, E., Ostrovsky, R., and Sahai, A. (2007). Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA. Association for Computing Machinery.

Kannwischer, M. J., Rijneveld, J., Schwabe, P., and Stoffelen, K. (2019). pqm4: Testing and benchmarking NIST PQC on ARM cortex-m4. Cryptology ePrint Archive, Paper 2019/844.

Kim, S., Ha, J., Son, M., Lee, B., Moon, D., Lee, J., Lee, S., Kwon, J., Cho, J., Yoon, H., and Lee, J. (2023). Aim: Symmetric primitive for shorter signatures with stronger security. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 401–415, New York, NY, USA. Association for Computing Machinery.

Kwon, J., Lee, S., Lee, B., Seo, H., and Cho, J. (2025). Efficient implementations of aimer post-quantum signature scheme for low-end to high-end iot devices. *IEEE Internet of Things Journal*, 12(22):46817–46837.

López, J. and Dahab, R. (2000). High-speed software multiplication in f2m. In Roy, B. and Okamoto, E., editors, *Progress in Cryptology —INDOCRYPT 2000*, pages 203–212, Berlin, Heidelberg. Springer Berlin Heidelberg.

Marsh, J. (2020). *Arm Helium Technology M-Profile Vector Extension (MVE) for Arm Cortex-M Processors Reference Book*. Arm Education Media.

NIST (2022). Post-Quantum Cryptography: Additional Digital Signature Schemes. `https://csrc.nist.gov/projects/pqc-dig-sig`.

Oliveira, T., Aranha, D. F., López, J., and Rodríguez-Henríquez, F. (2014). Fast point multiplication algorithms for binary elliptic curves with and without precomputation. In Joux, A. and Youssef, A., editors, *Selected Areas in Cryptography – SAC 2014*, pages 324–344, Cham. Springer International Publishing.

Weimerskirch, A., Stebila, D., and Shantz, S. C. (2003). Generic $gf(2^m)$ arithmetic in software and its application to ecc. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 79–92. Springer.

A. Helium MVE Instructions

Table 5 summarizes the Helium instructions used in this work; all operate on 128-bit registers.

Table 5. Helium MVE instructions used in this work.

Instruction	Intrinsic	Operation
VMULLB	vmullbq_poly_p16	Carry-less multiply of even 16-bit lanes: $(a_0b_0,\ a_2b_2,\ a_4b_4,\ a_6b_6)$, 32-bit results
VMULLT	vmulltq_poly_p16	Carry-less multiply of odd 16-bit lanes: $(a_1b_1,\ a_3b_3,\ a_5b_5,\ a_7b_7)$, 32-bit results
VEOR	veorq_u32	Bitwise XOR on all lanes: $(a_0 \oplus b_0,\ \ldots,\ a_{n-1} \oplus b_{n-1})$
VDUP	vdupq_n_u16	Broadcasts a scalar to all lanes: $(s,\ s,\ \ldots,\ s)$
VST1	vst1q_u16	Stores a 128-bit vector register to memory at address p: mem$[p] \leftarrow a$
VLD1	vld1q_u16	Loads a 128-bit vector from memory at address p: $a \leftarrow$ mem$[p]$
VREV32	vrev32q_u16	Reverses 16-bit lanes within each 32-bit group: $(a_1, a_0,\ a_3, a_2,\ a_5, a_4,\ a_7, a_6)$
VREV64	vrev64q_u16	Reverses 16-bit lanes within each 64-bit group: $(a_3, a_2, a_1, a_0,\ a_7, a_6, a_5, a_4)$
VPSEL	vpselq_u16	Selects lanes from a or b by predicate mask p: lane $i \leftarrow p_i\ ?\ a_i : b_i$
VSHL	vshlq_n_u32	Left-shifts each 32-bit lane by n bits: $(a_0 \ll n,\ \ldots,\ a_3 \ll n)$
VSHR	vshrq_n_u32	Right-shifts each 32-bit lane by n bits: $(a_0 \gg n,\ \ldots,\ a_3 \gg n)$
VGET	vgetq_lane_u32	Extracts a 32-bit lane k from a 128-bit register: lane$_k(a)$
VSET	vsetq_lane_u32	Inserts a 32-bit value into lane k of a 128-bit register
VCREATE	vcreateq_u16	Creates a 128-bit register from two 64-bit scalars: $(hi,\ lo)$

Types vary by operand width; suffixes `_u8`, `_u16`, `_u32` indicate the lane size used in each context.