

Efficient Vectorized Implementation of MQOM

David Santos¹, Julio López¹ and Francisco Rodríguez-Henríquez²

¹ Universidade Estadual de Campinas, Campinas, Brazil,
d261032@dac.unicamp.br, jlopez@unicamp.br

² Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
francisco.rodriguez@tii.ae

Abstract. Post-quantum signature schemes require efficient implementations to be viable in practice. MQOM, a third-round candidate in NIST’s additional call for digital signatures, is a promising target for SIMD optimization. This work leverages a transpose-enabled paired-matrix trick to develop a vectorized implementation of MQOM v2.1, corresponding to the Round-2 specification, with reduced wasted SIMD work and memory traffic. We evaluate MQOM v2.1 on Arm NEON, x86 AVX2, and x86 AVX-512, covering key generation, signing, verification, and memory; peak dynamic memory falls by up to 39.43%. We also include supplementary MQOM v3 experiments to assess how the matrix-layout optimization transfers to the Round-3 code.

Keywords: MQOM · Post Quantum Cryptography · Digital Signature Scheme

1 Introduction

Following Shor’s 1994 quantum algorithm [Sho94] and recent advancements that have lowered the estimated qubit requirements for factoring or computing discrete logarithms [BZG⁺26], a generalized concern has emerged that classical public-key cryptosystems (RSA and ECC) could eventually be efficiently broken by a powerful quantum computer. As a result, there is already great interest in coming up with alternatives that could be secure in that context.

In 2016, the United States National Institute of Standards and Technology (NIST) launched a standardization process to evaluate and select post-quantum algorithms and included digital signature schemes. By 2022, three signature schemes had been selected, ML-DSA [Nat24a], SLH-DSA [Nat24b] and FN-DSA [FHK⁺20]. As of now, NIST is conducting an additional call for digital signature schemes. NIST selected nine schemes for Round 3 in May 2026; HAWK later withdrew, leaving eight active candidates [Nat26].

Among these candidates, we focus on MQOM. Our main design and performance evaluation target MQOM v2.1 (Round 2) [BBFR25]. Since MQOM has advanced to v3.0, Section 6.6 provides supplementary experiments that test whether the matrix-layout optimization transfers to the Round-3 code [BBFR26]. MQOM’s security is based on the computational hardness of solving systems of equations over finite field extensions. Each equation has the form

$$y_i = x^T \cdot A_i \cdot x + b_i^T \cdot x,$$

where A_i is a square matrix, b_i and x are vectors of finite field elements and y_i is a field element. Considering properly chosen parameters, it should be hard for someone to find a solution x even if A_i , b_i and y_i are all public.

The solution of a system of equations can be used to create a set of n vector polynomials. These vectors would be distributed to n participants in a multi-party computation (MPC) setting. Here, however, following the MPC in the Head framework [IKOS07], the prover

holds all the shares, honestly executes the MPC scheme and commits to all the shares and interactions between participants. Later, the verifier chooses one share, and the prover opens all but that share, and the verifier, having the shares and the commitments, can check consistency and be convinced that the prover knows a witness up to some probability. In order to increase this probability, the protocol is repeated a sufficient number of times. This interactive protocol between two parties is known as a zero-knowledge proof and the Fiat-Shamir transform [FS87] is used in order to make it non-interactive and indeed a digital signature scheme.

In the context of the NIST competition, developing efficient software implementations of the candidates is crucial, as it leads to more conscientious, long-lasting industrial standards.

Contributions. With this goal in mind, this paper leverages SIMD instruction sets—including Arm NEON, x86 AVX2, and AVX-512—to improve the state-of-the-art performance of MQOM v2.1. Our contributions are as follows:

1. We propose a SIMD-aware paired-matrix layout and an efficient matrix multiplication algorithm, and provide a formal correctness proof for the underlying transpose-enabled transformations.
2. We present optimized MQOM v2.1 implementations for the ARM NEON, Intel AVX2, and Intel AVX-512 instruction sets, including an optimization of Gray-code folding.
3. We evaluate performance and memory across the three platforms and provide supplementary MQOM v3 experiments to assess portability of the matrix-layout optimization.

Organization. Section 2 reviews the components of MQOM that are most relevant to our implementation, including its MQ relation, MPC-in-the-Head structure, and the matrix operations that dominate the computation. Section 3 develops the transpose-enabled matrix-vector optimization and proves the transformations on which it relies. Section 4 then describes how we realize these optimizations with Arm NEON, AVX2, and AVX-512 instructions, together with the Gray-code folding optimization. Section 5 discusses the side-channel implications of the resulting SIMD implementations and motivates the row-wise variant used in our benchmarks. Section 6 evaluates performance and memory consumption across the three platforms and includes supplementary experiments on MQOM v3 to assess how well the matrix-layout optimization transfers to the revised scheme. Finally, Section 7 summarizes the main findings and discusses the remaining optimization opportunities.

2 MQOM background and computational structure

MQOM derives a digital signature scheme from a zero-knowledge proof of knowledge of a solution to a multivariate quadratic (MQ) system. We review here only the parts of the construction that are needed to understand the implementation bottlenecks addressed in the following sections; the complete protocol appears in the MQOM v2.1 specification [BBFR25].

2.1 MQ relation and MPC-in-the-Head structure

The public relation underlying MQOM consists of m quadratic equations over a finite field. For a secret witness $x \in F^n$, each equation has the form $y_i = x^T A_i x + b_i^T x$, where $A_i \in F^{n \times n}$ and $b_i \in F^n$. The Multivariate Quadratic problem asks an adversary to

recover such an x from the public system, and its concrete hardness has been studied extensively [BMSV22]. In MQOM, the matrices A_i are generated in lower-triangular form from a compact public seed, which avoids storing the full MQ instance in the public key.

MQOM uses MPC-in-the-Head [IKOS07] to turn knowledge of x into a signature. Since MQOM v2.0, its proof system has used threshold secret sharing within the Threshold-Computation-in-the-Head framework [FR25]. The prover secret-shares the witness, simulates the verification computation, and commits to the resulting views. A challenge determines which information is opened, allowing the verifier to check consistency without learning the witness. The Fiat–Shamir transform then derives this challenge from the transcript and signed message, yielding a non-interactive signature. Among NIST’s additional-signature candidates, SDitH and FAEST are related at the proof-system level: since Round 2, SDitH has followed the VOLE-in-the-Head framework [ABB⁺24], while FAEST likewise builds on VOLE-in-the-Head [BBdSG⁺23].

2.2 MQOM key generation, signing, and verification

This subsection summarizes the key-generation, signing, and verification procedures of the MQOM v2.1 specification [BBFR25]. We retain its notation and routine names to make the implementation discussion easier to follow.

Key generation. MQOM samples a master seed and derives from it both the secret solution x and a seed mseed_{eq} for the MQ equations. It expands mseed_{eq} into the public matrices and vectors $\{A_i, b_i\}$ and evaluates the system at x to obtain $y = (y_1, \dots, y_m)$. The public key contains the equation seed and y , while the secret key additionally contains x . Thus, the public key remains compact even though signing and verification can regenerate the complete MQ instance when needed.

Signing. The signer first regenerates $\{A_i, b_i\}$ from the public seed and initializes the commitment structure used by the threshold proof. The signing computation then evaluates the MQ relation on the shared witness across the protocol repetitions and forms the polynomial consistency information used by the proof. In the v2.1 specification, the routine `ComputePAlpha` performs this step and relies on `ComputePz`, where the dominant linear-algebra operations include products such as $A_i x_0[e]$ and $A_i x + b_i$. The signer hashes the commitments, polynomial data, public key, and message to obtain the Fiat–Shamir challenge, opens the challenged portions of the commitment, and serializes the resulting commitments, response data, and challenge nonce as the signature.

Verification. The verifier independently regenerates $\{A_i, b_i\}$ from the public key, reconstructs the Fiat–Shamir challenge, and checks the corresponding commitment openings. It then recomputes the missing polynomial-consistency data from the opened evaluations and the public MQ equations; in particular, `RecomputePAlpha` invokes `ComputePzEval`, whose main matrix operation has the form $A_i v_x + b_i r$. Finally, the verifier hashes the reconstructed polynomial data and accepts only when the resulting commitment matches the one contained in the signature. The verifier therefore checks consistency with the public MQ relation without recovering the secret witness.

2.3 Computational structure relevant to this work

Key generation, signing, and verification all operate on the same public matrices A_i , but they use them with different frequency. Key generation expands each matrix and uses it only once to compute the public MQ output, whereas signing and verification reuse the matrices throughout the proof computations and across protocol repetitions.

The lower-triangular structure of the A_i is central to the optimization in Section 3. In a conventional dense SIMD representation, roughly half of the matrix entries are zero, yet they still consume memory bandwidth and SIMD lanes. Since the matrices contain only public data, we can reorganize their layout and transpose selected matrices without

changing the MQ relation or introducing secret-dependent behavior. Section 3 develops these transformations, and Section 4 describes their implementation with NEON, AVX2, and AVX-512.

MQOM v2.1 replaces field-embedding batching with packing, adopts Gray-code ordering for the folding domain, and introduces a $GF(2^4)$ parameter set. These design changes affect both arithmetic and folding costs and directly motivate the folding optimization presented in Section 4. MQOM v3 subsequently revises the parameter sets and reworks parts of the matrix-multiplication and folding procedures, including features that overlap with our v2.1 optimizations. Section 6.6 examines v3 separately to assess the portability of the matrix-layout optimization [BBFR26]. Other recent studies focus on efficient MQOM implementations for embedded platforms [BF26] and on the security of the correlated GGM-tree commitment structure [FR26].

3 Transpose-enabled matrix-vector optimization

3.1 Transpose invariance

We now present and prove the following lemma, which enables the manipulation of the matrices used in MQOM while preserving the correctness of the scheme. This result underpins the optimizations introduced later, yielding reductions in both memory usage and execution time. We take as reference the specification of MQOM version 2.1 [BBFR25].

Definition 1. (Multivariate Quadratic Problem) Let $\mathbb{F}$ be a finite field, and m, n be positive integers. Let $[m]$ be the set of integers from 1 to m , inclusive. The Multivariate Quadratic (MQ) problem with parameters $(\mathbb{F}, m, n)$ is the following problem:

Let $(A_i)_{i \in [m]}, (b_i)_{i \in [m]}$, x and $y = (y_1, \dots, y_m)$ be such that:

1. x is uniformly sampled from $\mathbb{F}^n$,
2. for all $i \in [m]$, A_i is uniformly sampled from $\mathbb{F}^{n \times n}$,
3. for all $i \in [m]$, b_i is uniformly sampled from $\mathbb{F}^n$,
4. for all $i \in [m]$, y_i is defined as $y_i \leftarrow x^T A_i x + b_i^T x$.

From $(\{A_i\}, \{b_i\}, y)$, find x .

Lemma 1. For any instance of the Multivariate Quadratic Problem, replacing any matrix A_i with its transpose A_i^T does not change the resulting system of equations.

Proof. By Definition 1, $x \in \mathbb{F}^n$ and, for an arbitrary i , $A_i \in \mathbb{F}^{n \times n}$. Then, we have that $x^T A_i x \in \mathbb{F}^{1 \times 1}$. By the definition of matrix transpose, we have that:

$$(x^T A_i x) = (x^T A_i x)^T = (A_i x)^T (x^T)^T = x^T A_i^T x.$$

□

The matrices $\{A_i\}$ are used in three MQOM v2.1 procedures: KeyGen, ComputePz, and ComputePzEval, corresponding to Algorithms 1, 7, and 9 of [BBFR25]. Our transpose-enabled representation leaves the protocol unchanged and only replaces the matrix products involving A_i . Table 1 summarizes these substitutions, whose equivalence is proved below.

Table 1: Matrix operations modified by the transpose-enabled representation.

Procedure	Reference operation	Modified operation
KeyGen	$x^T A_i x$	$x^T A_i^T x$
ComputePz	$A_i x_0[e], A_i x + b_i$	$A_i^T x_0[e], A_i^T x + b_i$
ComputePzEval	$A_i v_x + b_i r$	$A_i^T v_x + b_i r$

The three procedures use the matrices in different contexts. KeyGen evaluates the quadratic forms that define the public MQ instance, whereas ComputePz and ComputePzEval perform the matrix-vector products required during signing and verification. In each case, we replace products involving A_i with the corresponding products using A_i^T , without changing the result of the computation. For KeyGen, Lemma 1 establishes the equivalence immediately. ComputePz and ComputePzEval involve additional bilinear terms, so we prove those substitutions directly below.

For KeyGen, Lemma 1 gives the required equality. In ComputePzEval (Algorithm 9 of [BBFR25]), substituting $v_t = A_i v_x + b_i r$ gives

$$v_{z,i} = v_x^T A_i^T v_x + r b_i^T v_x - y_i r^2.$$

Replacing A_i by A_i^T changes only the quadratic term, which is invariant by Lemma 1. In ComputePz (Algorithm 7 of [BBFR25]), write $u = x_0[e]$. The two output coefficients are

$$z_{0,i} = u^T A_i^T u, \quad z_{1,i} = u^T A_i^T x + x^T A_i^T u + b_i^T u.$$

The first is invariant by Lemma 1. For the second, the scalar identities $u^T A_i^T x = x^T A_i u$ and $x^T A_i^T u = u^T A_i x$ show that transposition exchanges the two bilinear terms and leaves their sum unchanged. These identities also hold over the extension field $\mathbb{K}$ used for u and v_x . Thus, all three procedures return the same outputs after the substitution.

3.2 Paired-matrix layout and vectorized multiplication

MQOM offers multiple parameter sets across security levels I , III , and V . For each parameter set, the number of bytes required to store the matrices A_i and vectors b_i is significant, such that using them as the scheme public-key would result in prohibitively large keys, which is undesirable. Instead, these elements are generated by expanding a seed and the seed forms the public key. The pseudo-random generator function (PRG) chosen to expand the key for category I is based on the Advanced Encryption Standard (AES) [Nat01] (Rijndael-256-256 for categories III and V) and, in order to generate extra bytes, it is necessary to make calls to the AES encryption function. This process, in turn, increases the execution time of the protocol. Thus, it is important to reduce the number of calls to the PRG function to the minimum possible, and as a result, one of the design choices made during the scheme proposal was to make all the matrices A_i lower triangular matrices.

One consequence of using lower triangular matrices is that approximately half of the storage allocated for each matrix is occupied by zero entries. Moreover, in implementations that use SIMD instruction sets to perform multiple field extension multiplications in parallel, a significant fraction of these multiplications are by zero. (See Figure 1).

Since the current matrix-vector multiplication method is suboptimal in both space complexity and computational cost, and because the matrices A_i contain only public data whose manipulation does not affect the scheme's security assumptions, we propose a new matrix-vector multiplication method, called the *Column-wise method*. The proposed method incorporates two optimizations. First, it computes $(x^T \cdot A_i) \cdot x$ instead of $x^T \cdot (A_i \cdot x)$. Second, by leveraging Lemma 1, it exploits a structural property of the set of matrices A_i $i \in [m]$ to further reduce the computational cost for an arbitrary vector $x \in \mathbb{F}^n$, as described below.

Column-wise matrix-vector multiplication

1. Let m be the number of matrices A_i , indexed from A_1 . We group the matrices into pairs $(A_1, A_2), \dots, (A_{m-1}, A_m)$, assuming m is even. If m is odd, the final matrix is handled separately.

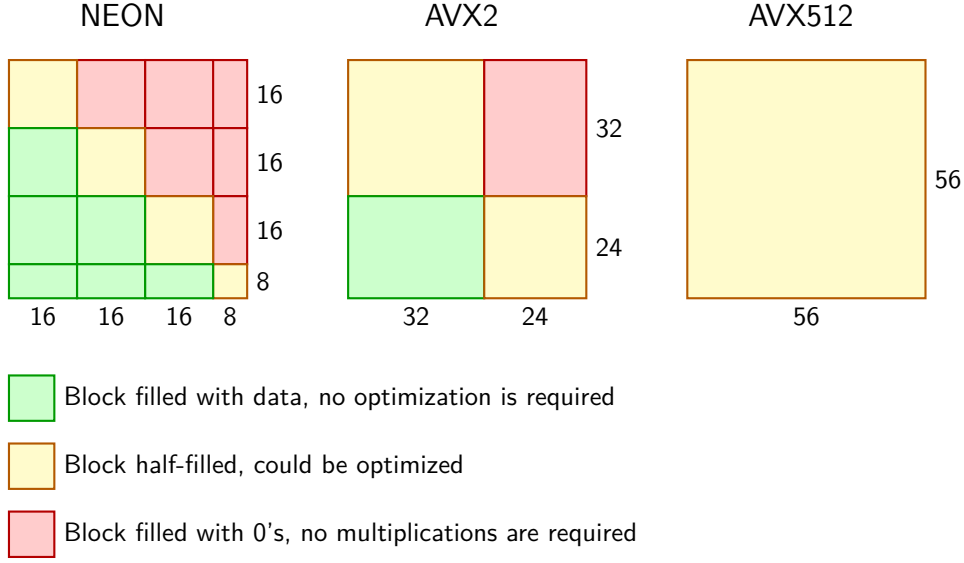


Figure 1: Comparison of the amount of multiplications by 0 for different SIMD instructions (and different registers sizes) for a fixed matrix dimension 56 x 56.

2. For each pair $P = (A_i, A_{i+1})$, we derive $P' = (A_i, A_{i+1}^T)$.
3. In this step we will copy the elements from the main diagonal of A_{i+1}^T to another vector d . From $P' = (A_i, A_{i+1}^T)$, we derive $P'' = (B_i, B_{i+1}, d)$, $B_i, B_{i+1} \in \mathbb{F}^{n \times n}$, $d \in \mathbb{F}^n$ such that

$$\begin{aligned}
 B_i &= A_i \\
 B_{i+1}[j][k] &= \begin{cases} A_{i+1}^T[j][k], & \text{if } j \neq k \\ 0, & \text{if } j = k \end{cases} \\
 d[j] &= A_{i+1}^T[j][j]
 \end{aligned}$$

4. From $P'' = (B_i, B_{i+1}, d)$ we derive $P''' = (C_i, d)$, where $C_i = B_i \oplus B_{i+1}$ (element-wise XOR, which corresponds to overlaying the matrices since B_i is a lower triangular matrix and B_{i+1} is an upper triangular matrix).
5. Let r denote the register size of the SIMD instructions being used. We partition C_i into a $c \times c$ grid of blocks, where $c = \lceil \frac{n}{r} \rceil$ (if $n \equiv 0 \pmod{r}$ the matrix is partitioned into uniform r blocks; otherwise, the boundary blocks will have smaller dimensions).
6. At this point, we have blocks comprised only of elements from B_i which we multiply by x and accumulate in a temporary variable $temp_i$, blocks comprised only of elements from B_{i+1} which we multiply and accumulate in a temporary variable $temp_{i+1}$ and mixed blocks which we multiply, separate the elements using masks and accumulate in the corresponding temporary variable.
7. Finally we compute the Hadamard product $d \circ x$ and accumulate in $temp_{i+1}$.

This transformation reduces the number of SIMD multiplications wasted on zero elements while also decreasing memory traffic. For example, let A be an $n \times n$ lower triangular matrix, such that each row i , $1 \leq i \leq n$ has i elements that could be not equal to 0. Let

r denote the register size of the target platform in bytes, allowing r multiplications over $GF(2^8)$ to be performed in parallel. Then, to multiply all the elements in row i , the number of parallelized multiplications we need is $\lceil \frac{i}{r} \rceil$ (twice as many are required for $GF(2^{16})$). As a result, the number of parallelized multiplications required to compute the product of two full matrices is:

$$2 \times \sum_{i=1}^n \left\lceil \frac{i}{r} \right\rceil.$$

On the other hand, with the new data layout, each row of a matrix pair contains n elements, and we require one additional vector containing n elements. Therefore, the total number of multiplications required for one pair is:

$$\sum_{i=1}^n \left\lceil \frac{n}{r} \right\rceil + \left\lceil \frac{n}{r} \right\rceil = (n+1) \times \left\lceil \frac{n}{r} \right\rceil.$$

In this new scenario, we introduce an additional overhead consisting of $\lceil \frac{n}{r} \rceil$ mask initializations (setting all bits in a register to 1), n mask updates (performing a right shift by one), n mask bit flips, and $2 \times n$ AND operations between the multiplication results and the masks for each pair of matrices. Overall, even on AVX-512 with GFNI, which provides the most efficient field multiplication among all evaluated implementations, the proposed transformation remains beneficial.

Considering the transformation from $x^T \cdot (A_i \cdot x)$ to $(x^T \cdot A_i) \cdot x$, we first distinguish between two types of operations. We use the term "addition" to refer to the XOR operation between two r -element vectors, and "sum" to refer to the XOR reduction of all elements within a single r -element vector. To compute $(A_i \cdot x)$, it is necessary to perform $\lceil \frac{n}{r} \rceil \times (n+1)$ loads to read the matrices from memory, $\lceil \frac{n}{r} \rceil \times (n+1)$ loads to read the vector x from memory $n+1$ times, and $2 \times n$ stores. Additionally, $2 \times n$ sums are required, corresponding to two sums for each row of the combined matrix. In contrast, for the transformed computation, we perform $\lceil \frac{n}{r} \rceil \times (n+1)$ loads to read the matrices from memory, $\lceil \frac{n}{r} \rceil \times \lceil \frac{n}{r} \rceil$ loads to read the vector x from memory $\lceil \frac{n}{r} \rceil$ times, $2 \times \lceil \frac{n}{r} \rceil$ stores. Moreover, no additional sums are required. However, this approach introduces an additional cost of $\lceil \frac{n}{r} \rceil \times n$ broadcasts, which remains beneficial overall. In summary, considering only the operations described above (additions are omitted since their cost is identical for all considered methods), the cost of multiplying two matrices using the reference method, the optimized method computing $x^T \cdot (A_i \cdot x)$ and the optimized method computing $(x^T \cdot A_i) \cdot x$ is given by:

Reference cost

$$2 \times \sum_{i=1}^n \left\lceil \frac{i}{r} \right\rceil (2 \times \text{loads} + \text{multiplications}) + 2 \times n (\text{sums} + \text{stores})$$

Optimized row-wise evaluation $x^T(A_i x)$ cost

$$\begin{aligned} & 2 \left\lceil \frac{n}{r} \right\rceil (n+1)(\text{loads}) + \left\lceil \frac{n}{r} \right\rceil (n+1)(\text{multiplications}) \\ & + 2n(\text{sums} + \text{stores}) + \left(4n + \left\lceil \frac{n}{r} \right\rceil \right) (\text{mask ops}) \end{aligned}$$

Optimized column-wise evaluation $(x^T A_i)x$ cost

$$\left\lceil \frac{n}{r} \right\rceil \times (n+1)(\text{loads} + \text{multiplications}) + \left\lceil \frac{n}{r} \right\rceil \times \left\lceil \frac{n}{r} \right\rceil (\text{loads}) + 2 \times \left\lceil \frac{n}{r} \right\rceil (\text{stores}) + (4 \times n + \left\lceil \frac{n}{r} \right\rceil)(\text{mask ops}) + \left\lceil \frac{n}{r} \right\rceil \times n(\text{broadcasts})$$

Algorithm 1 shows how to perform matrix-vector multiplication once the matrices are already combined in pairs. Both the elements of the matrix and the vector are in $GF(2^8)$. For a platform with registers of size r bytes, the columns are processed in batches of r columns. Then, we iterate row by row where there are three distinct scenarios we must cover:

1. The elements read all belong to the upper matrix (upper block, we use Algorithm 2);
2. A fraction of the elements belong to the upper matrix and a fraction of the elements belong to the lower matrix (mixed block, we use Algorithm 3);
3. The elements read all belong to the lower matrix (lower block, we use Algorithm 2).

There is also the case where the size of the matrix n is not divisible by r and we must deal with the last block with fewer elements separately. This case is covered in the if statements where the value of remainder is checked. In any case, the number of columns in a block is always less or equal than r .

Algorithms 2 and 3 give the helper routines used by Algorithm 1. In the pseudocode, $\vec{0}$ denotes an all-zero SIMD vector and $\vec{1}$ an all-one-bit mask; **broadcast** replicates one field element across the active SIMD lanes, while **mult** performs lane-wise field multiplication. In mixed diagonal blocks, the mask is shifted by one byte at each iteration to separate entries above and below the diagonal. The pseudocode describes the column-wise method; Section 5 presents the row-wise variant used in our benchmarks.

Algorithm 2 separated_block_multiply

Input: Matrix A of size $n \times n$; vector X ; block dimensions $b_{\text{row}}, b_{\text{column}}$; starting $\text{row}, \text{column}$

Output: Vector acc of size b_{column}

```

1: pos  $\leftarrow n \times \text{row} + \text{column}$ 
2: acc  $\leftarrow \vec{0}$ 
3: for k  $\leftarrow 0$  to  $b_{\text{row}} - 1$  do
4:   dups  $\leftarrow \text{broadcast}(X[\text{row} + k])$ 
5:   temp  $\leftarrow A[\text{pos} : \text{pos} + b_{\text{column}} - 1]$ 
6:   acc  $\leftarrow \text{acc} \oplus \text{mult}(\text{temp}, \text{dups})$ 
7:   pos  $\leftarrow \text{pos} + n$ 
8: end for
return acc
```

Algorithm 1 matrix_multiply

Input: Matrix A of size $n \times n$
 Vector d the removed diagonal
 Vector X of size n
 Register size r
 Vectors Y_{inf} Y_{sup} of size n to store the outputs

```

1: batches  $\leftarrow n/r$ 
2: remainder  $\leftarrow n - \text{batches} \times r$ 
3: for  $i \leftarrow 0$  to  $\text{batches} - 1$  do
4:    $\text{acc}_{\text{inf}} \leftarrow \vec{0}$   $\triangleright$  Vector with all bits set to 0
5:    $\text{acc}_{\text{sup}} \leftarrow \vec{0}$ 
6:   column  $\leftarrow i \times r$ 
7:   for  $j \leftarrow 0$  to  $i - 1$  do
8:     row  $\leftarrow r \times j$ 
9:      $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus \text{separated\_block\_multiply}(A, X, r, r, \text{row}, \text{column})$ 
10:  end for
11:  row  $\leftarrow r \times i$ 
12:   $\text{aux}_1, \text{aux}_2 \leftarrow \text{mixed\_block\_multiply}(A, d, X, r, r, \text{row}, \text{column})$ 
13:   $\text{acc}_{\text{inf}} \leftarrow \text{acc}_{\text{inf}} \oplus \text{aux}_1$ 
14:   $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus \text{aux}_2$ 
15:  for  $j \leftarrow i + 1$  to  $\text{batches} - 1$  do
16:    row  $\leftarrow r \times j$ 
17:     $\text{acc}_{\text{inf}} \leftarrow \text{acc}_{\text{inf}} \oplus \text{separated\_block\_multiply}(A, X, r, r, \text{row}, \text{column})$ 
18:  end for
19:  if remainder  $\neq 0$  then
20:    row  $\leftarrow r \times \text{batches}$ 
21:     $\text{acc}_{\text{inf}} \leftarrow \text{acc}_{\text{inf}} \oplus \text{separated\_block\_multiply}(A, X, \text{remainder}, r, \text{row}, \text{column})$ 
22:  end if
23:  store( $Y_{\text{inf}}[i \times r]$ ,  $\text{acc}_{\text{inf}}$ )
24:  store( $Y_{\text{sup}}[i \times r]$ ,  $\text{acc}_{\text{sup}}$ )
25: end for
26: if remainder  $\neq 0$  then
27:    $\text{acc}_{\text{inf}} \leftarrow \vec{0}$ 
28:    $\text{acc}_{\text{sup}} \leftarrow \vec{0}$ 
29:   column  $\leftarrow \text{batches} \times r$ 
30:   for  $i \leftarrow 0$  to  $\text{batches} - 1$  do
31:     row  $\leftarrow r \times i$ 
32:      $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus \text{separated\_block\_multiply}(A, X, r, \text{remainder}, \text{row}, \text{column})$ 
33:   end for
34:   row  $\leftarrow r \times \text{batches}$ 
35:    $\text{aux}_1, \text{aux}_2 \leftarrow \text{mixed\_block\_multiply}(A, d, X, \text{remainder}, \text{remainder}, \text{row}, \text{column})$ 
36:    $\text{acc}_{\text{inf}} \leftarrow \text{acc}_{\text{inf}} \oplus \text{aux}_1$ 
37:    $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus \text{aux}_2$ 
38:   store_incomplete( $Y_{\text{inf}}[\text{batches} \times r]$ ,  $\text{acc}_{\text{inf}}$ , remainder)
39:   store_incomplete( $Y_{\text{sup}}[\text{batches} \times r]$ ,  $\text{acc}_{\text{sup}}$ , remainder)
40: end if

```

Algorithm 3 mixed_block_multiply

Input: Matrix A, removed diagonal d, vector X, block dimensions $b_{\text{row}}, b_{\text{column}}$, starting row, column

Output: Vectors $\text{acc}_{\text{inf}}, \text{acc}_{\text{sup}}$

```

1: mask  $\leftarrow \vec{1}$ , pos  $\leftarrow n \times \text{row} + \text{column}$ 
2:  $\text{acc}_{\text{inf}} \leftarrow \vec{0}$ ,  $\text{acc}_{\text{sup}} \leftarrow \vec{0}$ 
3: for k  $\leftarrow 0$  to  $b_{\text{row}} - 1$  do
4:   mask  $\leftarrow \text{right\_shift}(\text{mask}, 1)$ 
5:   dups  $\leftarrow \text{broadcast}(X[\text{row} + k])$ 
6:   prod  $\leftarrow \text{mult}(A[\text{pos} : \text{pos} + b_{\text{column}} - 1], \text{dups})$ 
7:    $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus (\text{mask} \& \text{prod})$ 
8:    $\text{acc}_{\text{inf}} \leftarrow \text{acc}_{\text{inf}} \oplus (-\text{mask} \& \text{prod})$ 
9:   pos  $\leftarrow \text{pos} + n$ 
10: end for
11:  $\text{acc}_{\text{sup}} \leftarrow \text{acc}_{\text{sup}} \oplus \text{mult}(X[\text{row} : \text{row} + b_{\text{row}} - 1], d[\text{row} : \text{row} + b_{\text{row}} - 1])$ 
return  $\text{acc}_{\text{inf}}, \text{acc}_{\text{sup}}$ 

```

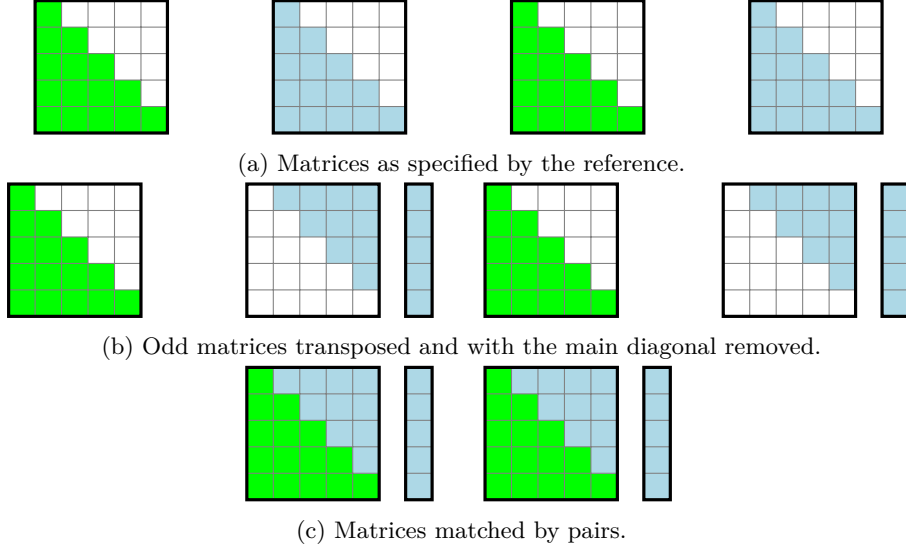


Figure 2: Steps taken to optimize matrix multiplication in MQOM.

4 Implementation details

The purpose of this section is to provide a more detailed description of the platforms studied in this work, as well as the instructions used to implement the proposed optimizations. Specifically, we examine the two additional steps introduced in the updated algorithm for each platform, namely:

- Duplicating elements from a vector to multiply by all the columns in the matrix (consequence of doing $(x^T \cdot A) \cdot x$ instead of $x^T \cdot (A \cdot x)$).
- Updating the mask used to separate the result of a multiplication (consequence of combining pairs of matrices).

In this section, we describe how each step of the proposed method is implemented using SIMD instructions on the architectures considered in this work.

4.1 NEON

NEON is an instruction set that is implemented in a subset of Arm processors. For example, all processors with Armv8-A should be able to utilize NEON instructions. NEON capable processors also come equipped with 32 128-bit registers and are able to interpret these registers as vectors, with either 64, 32, 16 or 8-bit elements. The instructions allow to perform operations on vectors stored on these registers which range from arithmetic operations, logical operations, memory related operations to matrix transposition, polynomial multiplication and AES specific instructions. Among them, we have the instruction `vextq_u8` (Figure 3) which receives as parameters two vectors of 16 8-bit unsigned integers, concatenates them and selects 16 8-bit unsigned integers from the result, starting from an offset.

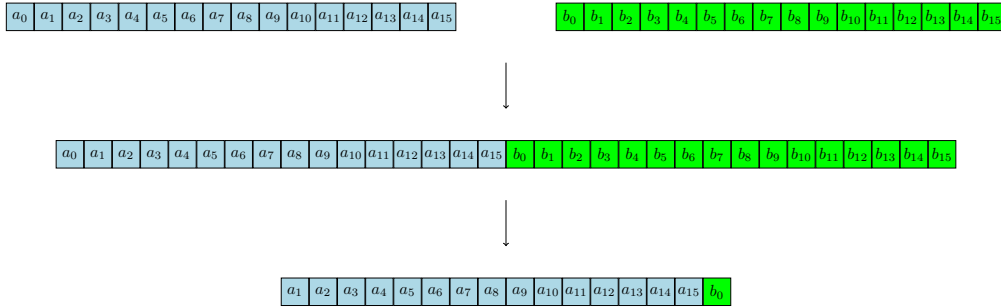


Figure 3: Result of calling the `vextq_u8` NEON instruction with vectors $\vec{a}$ and $\vec{b}$ and an offset of 1.

The instruction `vdupq_laneq_u8` (Figure 4) receives as parameter one vector of 16 8-bit unsigned integers and a position in this vector, then creates copies of the corresponding elements in all the positions of the output vector.

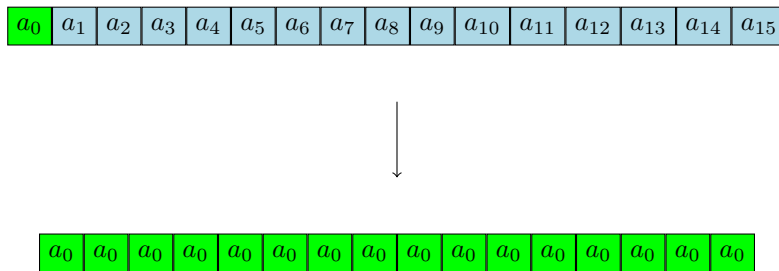


Figure 4: Result of calling the `vdupq_laneq_u8` NEON instruction with vector $\vec{a}$ and immediate equal to 0.

An important thing to notice is that since `vdupq_laneq_u8` uses an immediate as the second parameter, the value should be known in compilation time. As a consequence, we had to break the broadcast procedure in two steps. First we always called `vdupq_laneq_u8` with immediate equal to 0. This way the element in the first position in the register would be copied in each iteration. Then, we called `vextq_u8` with two copies of x and offset equal to 15 to put the next element in the first position of x . In order to have a suitable

mask at each iteration, we started with the mask equal to $\vec{1}$ and keep a vector aux with value equal to $\vec{0}$. Then, at each iteration, make $\text{mask} \leftarrow \text{vextq_u8}(\text{aux}, \text{mask}, 15)$.

4.2 AVX2/AVX512

Both AVX2 and AVX-512 are SIMD instruction sets implemented in some Intel x86 processors. They provide vector registers: AVX2 features 16 256-bit YMM registers, while AVX-512 provides 32 512-bit ZMM registers. In both cases, the registers are partitioned into 128-bit lanes. The instructions which operate on the values stored in these registers are analogous to the ones in NEON covering arithmetic operations, logic operations, memory access, AES specific instructions and many others. Among the AVX2 instructions there is the `_mm256_alignr_epi8` instruction (Figure 5) that has a behavior similar to `vextq_u8` but slightly different. The design choice, then, was to treat each 256-bit register as two 128-bit lanes. In order to create copies of all elements in a vector x we used both `_mm_alignr_epi8` and `_mm256_broadcastb_epi8`, very similarly to the NEON code. Then, in order to update the mask, the instruction, `_mm256_alignr_epi8` was used in the following manner, we start with mask equal to $\vec{1}$, and for the first lane we have aux with the first 128 bits equal to 1 and the last 128 bits equal to 0, we update calling $\text{mask} \leftarrow \text{_mm256_alignr_epi8}(\text{aux}, \text{mask}, 15)$. For the second lane we make aux equal to $\vec{0}$ and use the same call to update mask.

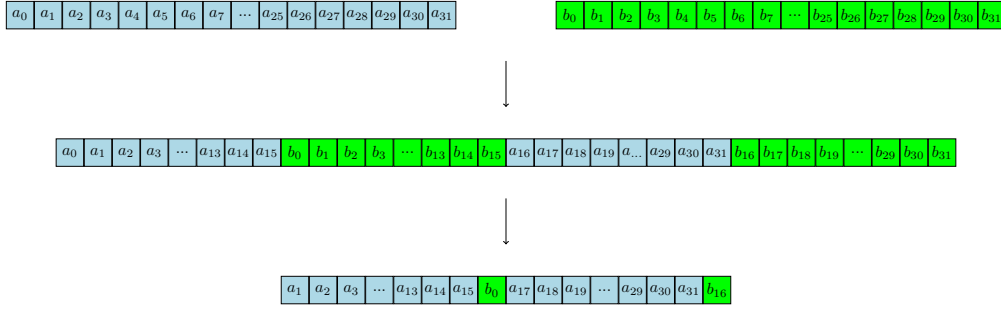


Figure 5: Result of calling the `_mm256_alignr_epi8` AVX2 instruction with vectors $\vec{a}$ and $\vec{b}$ and an offset of 1.

In AVX512, we still needed both `_mm_alignr_epi8` and `_mm512_broadcastb_epi8` (same as `_mm256_broadcastb_epi8` but for 512 bits) to create copies of all elements in a vector x , but we could use the instruction `_mm512_mask_blend_epi8` and a `_mmask64` control variable to get around the difficulty of operating between 128-bit lanes.

4.3 Gray-code folding optimization

MQOM v2.1 orders the folding domain using Gray code. At each iteration, the reference implementation recomputes consecutive Gray codes and searches for the bit position that changes, which determines the accumulator index p_i . We compute this index directly as

$$p_i = \begin{cases} 0, & i \text{ even,} \\ 31 - \text{clz}(i \oplus j), & i \text{ odd,} \end{cases} \quad j = (i + 1) \bmod N.$$

Here, N is the power-of-two domain size and `clz` counts leading zeros in a 32-bit word. The wraparound $j = 0$ for $i = N - 1$ implements the convention $\omega_N = 0$ from the specification. For odd i , the position of the highest set bit of $i \oplus (i + 1)$ gives p_i , while for even i the changed position is always 0. This closed form eliminates both Gray-code recomputation

and the bit-by-bit search. Separating even and odd iterations also gives the compiler a fixed accumulator index on every even iteration. Section 6.5 compares the resulting effect on signing and verification.

5 Side-channel considerations

Since cryptographic schemes are typically based on mathematical problems that are computationally infeasible to solve within a practical timeframe, attackers have also explored alternative methods for extracting secret information. These techniques, collectively known as Side-Channel Attacks (SCAs), exploit information leaked during the implementation or execution of a cryptographic algorithm rather than weaknesses in the underlying mathematics. Among the various types of SCAs, we focus on two categories that are particularly relevant to our implementation.

The first category comprises timing attacks, which exploit variations in execution time that arise when a program processes different secret inputs. Attacks such as [BB03] can even be carried out remotely and independently of the underlying platform, making them particularly powerful and highlighting the importance of mitigating timing-based information leakage whenever possible. In the context of a high-level implementation, the primary sources of concern are conditional branches and loops whose behavior depends on secret data. As shown in Algorithm 3, however, the loop bounds and conditional statements depend only on the matrix dimensions and the platform’s register size, both of which are public parameters. Consequently, the control flow is independent of secret data. Nevertheless, we do not claim that the implementation provides comprehensive resistance against side-channel attacks.

The second category comprises electromagnetic (EM) side-channel attacks, which exploit the electromagnetic radiation emitted by a device during the execution of cryptographic operations. Although these attacks are generally less practical than timing attacks, as they require physical proximity and specialized measurement equipment, they can still compromise secret information. For instance, [AYW⁺21] demonstrated the extraction of secret keys from smartphones under controlled experimental conditions. Despite the technical challenges posed by the complexity and high clock frequencies of modern smartphones, these results demonstrate the practical feasibility of EM side-channel analysis against real-world devices.

In this context, EM side-channel attacks targeting SIMD implementations remain relatively unexplored, although they may become increasingly relevant as SIMD-accelerated cryptographic implementations become more widespread. For the purposes of this work, however, EM side-channel analysis is considered out of scope, as we were unable to measure electromagnetic emissions on the evaluated devices.

Nevertheless, based on leakage models commonly used in EM side-channel analysis of non-SIMD implementations—such as those based on the Hamming weight and Hamming distance of intermediate values—it is reasonable to consider the potential implications of our implementation. In particular, some multiplication operations involve secret-key data, and broadcasting secret-key elements across SIMD lanes to perform multiplications with all matrix columns could increase EM leakage. This concern is particularly relevant for the $GF(2)$ variants. Broadcasting a secret bit across the SIMD lanes produces either an all-zero or an all-one register. Under Hamming-weight or Hamming-distance leakage models, this highly regular pattern may be easier to distinguish than a register containing different secret-vector elements. We emphasize that this observation motivates our leakage model rather than demonstrating a practical attack. It also motivates the row-wise formulation introduced below, which avoids broadcasting secret-dependent values.

To mitigate potential EM side-channel attacks, we propose an alternative approach for computing the sequence of matrix-vector multiplications required by the MQOM scheme,

which we refer to as the *Row-wise method*. The main idea is to avoid broadcasting secret-dependent values while preserving the timing-side-channel properties discussed earlier in this section. Specifically, instead of the original computation, we evaluate $x^T \cdot (A_i \cdot x)$ using the sequence of operations described below, which closely follows the methodology presented in Section 3.

Row-wise matrix-vector multiplication

1. Let m be the number of matrices A_i , indexed from A_1 . We group the matrices into pairs $(A_1, A_2), \dots, (A_{m-1}, A_m)$, assuming m is even. If m is odd, the final matrix is handled separately.
2. For each pair $P = (A_i, A_{i+1})$, we derive $P' = (A_i, A_{i+1}^T)$.
3. From $P' = (A_i, A_{i+1}^T)$, we derive $P'' = (B_i, B_{i+1})$, $B_i, B_{i+1} \in \mathbb{F}^{(n+1) \times n}$ such that

$$B_i = \begin{bmatrix} \vec{0} \\ A_i \end{bmatrix}$$

$$B_{i+1} = \begin{bmatrix} A_{i+1}^T \\ \vec{0} \end{bmatrix}$$

4. From $P'' = (B_i, B_{i+1})$ we derive C_i , where $C_i = B_i \oplus B_{i+1}$ (element-wise XOR).
5. We process C_i row by row, r elements at a time, where r is the register width.
6. Finally, we add the r remaining elements resulting in a single element which is stored in memory.

Compared to our first approach, the second approach introduces an overhead in the final step, where the r elements still need to be accumulated (using XOR operations) and the results stored in memory individually. To mitigate this overhead, we multiply four rows simultaneously up to Step 5 and use matrix transposition intrinsics (NEON) or shuffle instructions (AVX2/AVX-512) to perform the final accumulation of the four rows in parallel and store the four results at once. The benchmark results reported in Section 6 use the row-wise variant.

6 Evaluation

Following [BF26], we report timing measurements primarily for parameter sets over $GF(2^4)$. Level-I comparisons across $GF(2)$, $GF(2^4)$, and $GF(2^8)$ are also included in the artifact:

https://github.com/deividafonso281/vectorized_mqom

We base our implementation and benchmark harness on MQOM v2.1, and all optimized variants pass the official known-answer tests. The supplementary artifact provides the complete source code, build configuration, and benchmark-reproduction instructions. We compile all measurements with GCC 11.4.0 using the flags recorded in the artifact Makefiles. Apart from the NEON cycle-counter adaptation described below, we retain the MQOM v2.1 benchmark harness unchanged. The reported end-to-end key-generation, signing, and verification timings include the cost of matrix packing.

We also adopt the alignment strategy of the MQOM v2.1 implementation and do not instrument cache misses separately. Instead, we evaluate the proposed layouts through their end-to-end cycle counts, which capture the effect of reducing load/store operations and unnecessary SIMD multiplications.

Throughout this section, O/R denotes Ours/Reference. The suffix “r3” refers to the three-round MQOM v2.1 variant and should not be confused with MQOM v3 [BBFR26].

6.1 Platforms

Because MQOM v2.1 provides no NEON-optimized reference implementation, we derive our NEON baseline from the official AVX2 code. The port uses NEON AES instructions, the XKCP SHAKE implementation ¹, and finite-field multiplication similar to [OFRL25]. To avoid the secret-dependent broadcast pattern discussed in Section 5, we benchmark the row-wise variant. We collect all NEON measurements on a Raspberry Pi 5 (Cortex-A76) at 2.4 GHz using PMCCNTR_EL0, with 5,000 executions per result.

For AVX2 and AVX-512, we use the MQOM v2.1 reference implementation as the baseline and benchmark both variants on an 11th Gen Intel Core i7-1165G7 at 2.80 GHz, with HyperThreading and TurboBoost disabled. Each result again averages 5,000 executions, and we report timings in millions of cycles. The AVX-512 implementation uses GFNI to accelerate finite-field multiplication; because multiplication becomes cheaper, the matrix optimization accounts for a smaller fraction of the total execution time and yields a smaller relative speedup.

6.2 Comparison between variants

We compare Level-I performance across key generation (Figure 6), signing (Figure 7), and verification (Figure 8) for $GF(2)$, $GF(2^4)$, and $GF(2^8)$; negative values indicate a slowdown. A few key-generation configurations perform worse because key generation does not use the folding phase and multiplies each expanded matrix only once. As a result, it has fewer opportunities than signing and verification to amortize the one-time cost of packing and transposing each matrix. In these configurations, this setup overhead can exceed the savings from the optimized multiplication. When the computation reuses the matrices, however, the optimization remains beneficial.

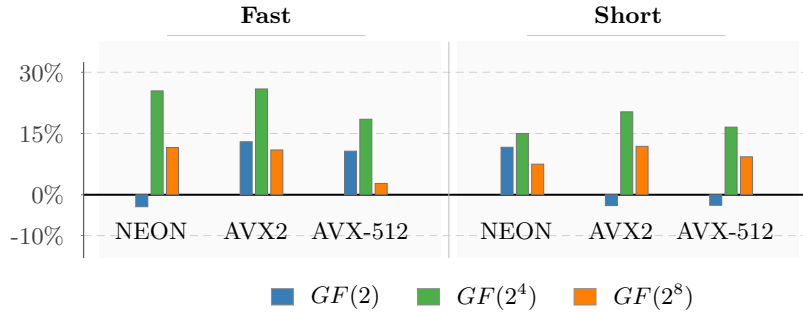


Figure 6: Percentage reduction in key generation time achieved by the proposed optimization for different variants across multiple platforms.

¹<https://github.com/XKCP/XKCP>

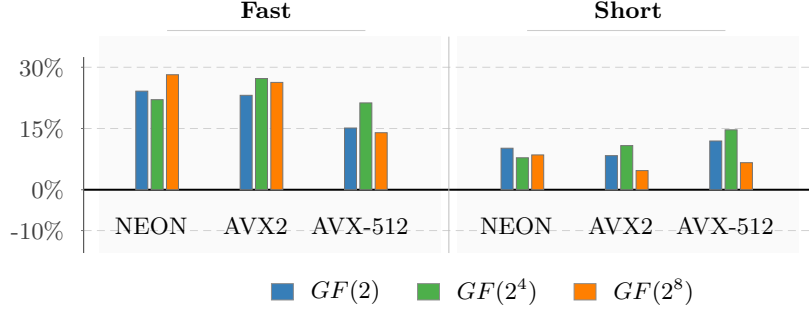


Figure 7: Percentage reduction in signature generation time achieved by the proposed optimization for different variants across multiple platforms.

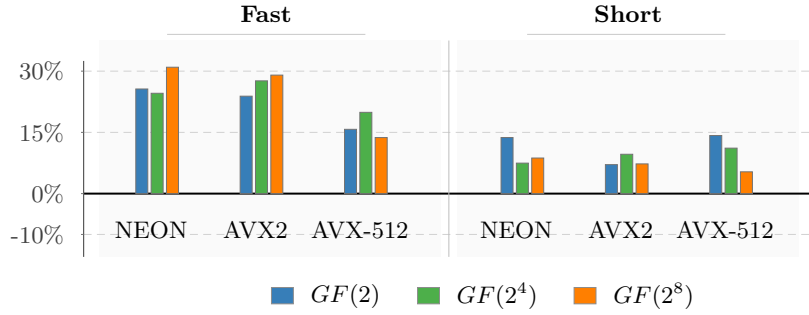


Figure 8: Percentage reduction in verification time achieved by the proposed optimization for different variants across multiple platforms.

6.3 End-to-end timings

Table 2: End-to-end MQOM v2.1 timings over $GF(2^4)$, in millions of cycles. Bold marks the lower (better) value in each Ours/Reference pair; ties are left unbolded.

Platform	Level	Variant	Key Gen		Sign		Verify	
			Ours	Ref.	Ours	Ref.	Ours	Ref.
NEON	I	fast	0.49	0.66	5.55	7.11	4.17	5.52
		short	0.50	0.58	18.58	20.16	17.91	19.36
	III	fast	2.15	2.55	19.79	24.72	18.99	23.32
		short	2.26	2.52	87.36	91.82	77.40	81.53
	V	fast	4.05	4.99	37.36	52.03	35.39	48.72
		short	4.26	4.98	136.36	150.03	133.46	144.94
AVX2	I	fast	0.32	0.43	2.69	3.70	2.26	3.12
		short	0.39	0.49	7.92	8.88	7.71	8.54
	III	fast	1.38	1.71	10.93	13.93	10.49	13.26
		short	1.38	1.54	39.51	40.86	36.63	37.53
	V	fast	2.84	3.47	23.97	31.54	22.67	29.45
		short	2.65	3.21	74.35	77.92	71.32	73.90
AVX-512+GFNI	I	fast	0.28	0.34	1.66	2.11	1.40	1.75
		short	0.35	0.41	5.61	6.57	5.57	6.27
	III	fast	1.29	1.48	7.08	8.36	6.99	7.99
		short	1.28	1.31	29.59	32.31	27.58	31.06
	V	fast	2.55	2.89	12.05	14.67	11.41	13.85
		short	2.33	2.62	46.00	50.72	43.54	47.97

Table 2 shows consistent improvements across all tested configurations, with the largest relative gains generally appearing in the fast variants. For example, on AVX2 at Level I, the fast signing time is reduced by 27.3%, compared with 10.8% for the short variant. The component profiles in Tables 4 and 5 help explain this difference. Matrix multiplication represents a larger fraction of the running time for the fast instances, whereas Encryption and XOF account for more of the cost in the short variants. Improvements to matrix multiplication thus have a greater impact on the overall running time of the fast instances. Key generation shows smaller gains because each expanded matrix is used only once, leaving less opportunity to amortize the one-time packing and transposition cost than in signing and verification.

6.4 Peak dynamic memory

Because the row-wise and column-wise variants differ by less than 1% in peak dynamic allocation, we report only the row-wise results. AVX2 and AVX-512 use the same amount of dynamic memory, so Table 3 combines them in a single column. Across all measured configurations, our optimization reduces peak dynamic memory by as much as 39.43%.

Table 3: Peak dynamic memory for MQOM v2.1 over $GF(2^4)$. Bold marks the lower (better) value in each Ours/Reference pair; ties are left unbolded.

Platform	Level	Variant	Peak dynamic memory (KB)	
			Ours	Ref.
NEON	I	fast r3	152.62	195.74
		short r3	257.98	301.10
	III	fast r3	326.92	458.80
		short r3	492.67	616.72
	V	fast r3	655.65	1015.02
		short r3	898.02	1242.86
AVX2/AVX-512	I	fast r3	102.70	145.82
		short r3	158.62	201.74
	III	fast r3	253.48	385.36
		short r3	346.27	470.32
	V	fast r3	551.97	911.34
		short r3	690.66	1035.50

6.5 Component-level profiles

To identify where the end-to-end gains arise, we profile Encryption, XOF, MatrixMult, and Fold separately during signing and verification. MatrixMult captures the contribution of the matrix-layout optimization, while Fold captures the Gray-code optimization; Encryption and XOF provide control components. Because both profiles use the same component groups and units, we can compare signing and verification directly.

Table 4: Signature-generation profile for MQOM v2.1 over $GF(2^4)$, in millions of cycles. Bold marks the lower (better) value in each Ours/Reference pair; ties are left unbolded.

Platform	Level	Variant	Encryption		XOF		MatrixMult		Fold	
			Ours	Ref.	Ours	Ref.	Ours	Ref.	Ours	Ref.
NEON	I	fast r3	1.01	1.01	2.93	2.93	0.69	1.89	0.06	0.12
		short r3	5.51	5.53	10.54	10.53	1.11	1.95	0.24	0.63
	III	fast r3	7.72	7.74	5.23	5.23	3.63	7.71	0.12	0.26
		short r3	40.43	40.59	36.51	36.50	5.93	9.02	0.55	1.26
	V	fast r3	10.34	10.34	9.28	9.28	11.50	24.43	0.23	0.42
		short r3	56.00	56.25	51.08	51.05	20.93	30.67	1.13	2.17
AVX2	I	fast r3	0.55	0.56	0.83	0.83	0.70	1.45	0.08	0.15
		short r3	2.80	2.97	2.65	2.65	1.34	1.58	0.33	0.68
	III	fast r3	3.77	3.76	1.51	1.51	3.53	6.03	0.14	0.22
		short r3	19.34	19.33	10.58	10.76	6.75	7.20	0.52	0.93
	V	fast r3	6.02	6.13	2.30	2.31	11.35	17.79	0.19	0.29
		short r3	32.22	33.12	13.29	13.29	23.38	24.81	0.74	1.35
AVX-512+GFNI	I	fast r3	0.54	0.59	0.54	0.54	0.14	0.30	0.05	0.16
		short r3	3.04	3.18	1.63	1.64	0.10	0.19	0.17	0.73
	III	fast r3	3.87	3.88	0.91	0.92	0.63	1.08	0.11	0.62
		short r3	19.87	19.81	6.71	6.69	0.48	0.73	0.73	3.04
	V	fast r3	5.97	6.06	1.40	1.40	1.52	2.85	0.12	0.80
		short r3	32.31	32.79	8.02	8.01	1.47	2.27	0.91	3.88

Table 5: Verification profile for MQOM v2.1 over $GF(2^4)$, in millions of cycles. Bold marks the lower (better) value in each Ours/Reference pair; ties are left unbolded.

Platform	Level	Variant	Encryption		XOF		MatrixMult		Fold	
			Ours	Ref.	Ours	Ref.	Ours	Ref.	Ours	Ref.
NEON	I	fast	1.02	1.02	1.78	1.78	0.69	1.93	0.06	0.13
		short	5.54	5.54	9.97	9.96	1.11	1.94	0.24	0.67
	III	fast	7.68	7.69	5.21	5.21	3.62	7.73	0.12	0.28
		short	40.26	40.28	27.43	27.44	5.92	8.89	0.54	1.39
	V	fast	10.06	10.05	9.23	9.24	11.49	24.51	0.24	0.45
		short	54.83	54.80	50.88	50.86	20.89	30.39	1.03	2.19
AVX2	I	fast	0.60	0.59	0.48	0.48	0.70	1.48	0.08	0.14
		short	2.92	3.05	2.46	2.46	1.34	1.56	0.34	0.68
	III	fast	3.78	3.78	1.50	1.50	3.53	6.08	0.14	0.25
		short	19.62	19.29	7.82	7.82	6.74	7.36	0.52	1.05
	V	fast	5.68	5.70	2.29	2.29	11.34	17.83	0.20	0.33
		short	30.19	30.69	13.22	13.21	23.32	24.53	0.75	1.50
AVX-512+GFNI	I	fast	0.57	0.63	0.30	0.30	0.14	0.30	0.05	0.15
		short	3.21	3.19	1.50	1.51	0.10	0.19	0.17	0.67
	III	fast	3.91	3.90	0.91	0.91	0.62	1.08	0.11	0.57
		short	20.07	20.14	4.70	4.69	0.47	0.72	0.72	3.78
	V	fast	5.59	5.71	1.39	1.39	1.49	2.81	0.13	1.00
		short	30.26	30.48	7.97	7.97	1.45	2.21	0.92	4.21

Matrix multiplication is among the most expensive components and accounts for a substantial share of the cycles on NEON and AVX2, where our optimization delivers the largest component-level gains. On AVX-512, GFNI makes field multiplication substantially cheaper, reducing the relative benefit of our optimization.

The folding optimization of Section 4.3 removes repeated Gray-code index computations and gives the compiler a fixed accumulator index on even iterations. This produces a clear

improvement in the signing profiles. For example, on AVX2 at Level I, the Fold component for the short variant decreases from 0.68 to 0.33 million cycles in Table 4.

The verification measurements show a different behavior. For the same AVX2 Level I short instance, the reference Fold component requires only 0.04 million cycles, whereas our measured implementation takes 0.34 million cycles. Although the optimization reduces the work needed to determine the Gray-code index, this reduction does not translate into a lower overall folding cost during verification. The component profiles do not isolate the source of this additional cost, so we do not attribute the slowdown to a specific operation. The end-to-end verification gains in Table 2 instead come from the matrix-multiplication optimization.

6.6 Supplementary validation on MQOM v3

MQOM v3.0 was released for Round 3 in August 2026 [BBFR26]. It removes the $GF(2^8)$ and “*-r3” instances, adds $GF(2)$ shorter variants and correlated-tree/one-tree variants, and changes the $GF(2^4)$ parameters. The v3 implementation also incorporates batching and folding changes that overlap with parts of our v2.1 optimization.

We use v3 only to assess the portability of our optimizations. For fast variants, we reuse the v2.1 row-wise multiplication. For short/shorter variants, we adapt the v3 batched-column implementation while preserving its precomputation. We do not modify v3 folding, which already reduces Gray-code work and batches seed expansion.

Tables 6–8 report all v3 measurements below. Timings are in millions of cycles, and CT and OT denote the correlated-tree and one-tree variants, respectively. The results are mixed: NEON and AVX-512 show substantial savings for several $GF(2^4)$ fast instances, whereas AVX2 includes neutral cases and regressions, notably at Level V. This is consistent with v3 already incorporating related batching and precomputation, which can reduce or eliminate the benefit of the additional matrix-layout transformation in some configurations.

Table 6: MQOM v3 benchmarks on NEON (millions of cycles; lower is better; bold marks the lower Ours/Reference value).

Level	Variant	Tree	Key Gen		Sign		Verify	
			Ours	Ref.	Ours	Ref.	Ours	Ref.
I	$GF(2)$ shorter	CT	0.38	0.49	23.84	24.83	15.53	15.77
		OT	0.38	0.49	42.94	43.40	40.01	40.51
	$GF(2^4)$ fast	CT	0.21	0.23	2.37	3.03	2.19	2.79
		OT	0.21	0.23	2.86	3.48	2.65	3.25
	$GF(2^4)$ short	CT	0.18	0.18	7.67	7.86	7.29	7.47
		OT	0.18	0.18	10.19	10.41	9.97	10.22
III	$GF(2)$ shorter	CT	2.74	3.10	84.31	86.07	70.88	72.08
		OT	2.75	3.29	113.19	114.91	100.67	101.32
	$GF(2^4)$ fast	CT	0.99	1.02	10.59	13.07	9.84	12.10
		OT	0.99	1.02	12.46	15.13	11.61	13.96
	$GF(2^4)$ short	CT	0.94	0.94	37.72	38.51	33.37	33.72
		OT	0.94	0.94	53.85	54.37	48.61	49.06
V	$GF(2)$ shorter	CT	5.09	10.52	159.89	168.18	143.75	151.77
		OT	4.94	10.97	190.85	201.06	179.51	188.89
	$GF(2^4)$ fast	CT	1.75	1.86	22.45	28.50	20.86	26.45
		OT	1.74	1.88	27.25	34.74	23.81	30.22
	$GF(2^4)$ short	CT	1.69	1.73	67.20	68.59	59.53	61.09
		OT	1.70	1.73	83.43	84.87	79.19	79.97

Table 7: MQOM v3 benchmarks on AVX2+VAES (millions of cycles; lower is better; bold marks the lower Ours/Reference value).

Level	Variant	Tree	Key Gen		Sign		Verify	
			Ours	Ref.	Ours	Ref.	Ours	Ref.
I	$GF(2)$ shorter	CT	0.30	0.38	18.72	18.95	11.27	11.42
		OT	0.30	0.39	34.03	34.58	32.52	32.61
	$GF(2^4)$ fast	CT	0.18	0.16	2.10	2.04	1.93	2.03
		OT	0.18	0.14	2.79	2.80	2.58	2.62
	$GF(2^4)$ short	CT	0.18	0.20	5.45	5.70	5.04	5.26
		OT	0.18	0.20	9.15	9.35	8.78	8.96
III	$GF(2)$ shorter	CT	1.85	2.16	52.64	54.51	46.42	47.92
		OT	1.85	2.16	88.06	90.32	83.28	84.58
	$GF(2^4)$ fast	CT	1.01	1.04	9.53	9.01	9.04	8.96
		OT	1.01	1.04	12.79	12.88	12.09	12.19
	$GF(2^4)$ short	CT	0.71	0.74	25.03	25.55	22.98	23.81
		OT	0.71	0.74	45.32	45.92	43.15	44.06
V	$GF(2)$ shorter	CT	3.27	3.92	95.27	101.26	94.78	99.13
		OT	3.28	3.93	140.11	145.16	135.85	140.87
	$GF(2^4)$ fast	CT	1.83	1.79	21.41	19.82	20.44	18.95
		OT	1.83	1.79	27.69	25.51	24.84	22.84
	$GF(2^4)$ short	CT	1.29	1.31	41.15	43.33	39.02	41.41
		OT	1.29	1.32	65.09	67.06	63.55	66.22

Table 8: MQOM v3 benchmarks on AVX-512+VAES+GFNI (millions of cycles; lower is better; bold marks the lower Ours/Reference value).

Level	Variant	Tree	Key Gen		Sign		Verify	
			Ours	Ref.	Ours	Ref.	Ours	Ref.
I	$GF(2)$ shorter	CT	0.28	0.35	14.49	14.89	6.90	6.98
		OT	0.28	0.35	17.07	17.25	15.65	16.06
	$GF(2^4)$ fast	CT	0.10	0.12	0.97	1.33	0.92	1.22
		OT	0.10	0.10	1.13	1.49	1.04	1.32
	$GF(2^4)$ short	CT	0.13	0.16	3.70	3.75	3.40	3.44
		OT	0.13	0.16	4.35	4.39	4.05	4.09
III	$GF(2)$ shorter	CT	1.74	1.93	36.42	36.34	30.26	30.60
		OT	1.74	1.93	47.74	46.20	41.54	41.58
	$GF(2^4)$ fast	CT	0.80	0.91	4.91	5.94	4.56	5.72
		OT	0.80	0.91	5.58	6.74	5.28	6.46
	$GF(2^4)$ short	CT	0.54	0.60	18.13	17.95	16.33	16.44
		OT	0.54	0.60	23.94	24.01	22.31	22.41
V	$GF(2)$ shorter	CT	3.03	3.34	49.61	50.85	44.46	45.80
		OT	3.03	3.33	61.84	63.02	56.44	57.58
	$GF(2^4)$ fast	CT	1.39	1.53	7.53	10.31	7.00	9.50
		OT	1.39	1.54	10.08	12.94	8.09	10.70
	$GF(2^4)$ short	CT	0.90	1.01	23.16	23.50	21.02	21.36
		OT	0.90	1.00	28.93	29.32	27.65	27.86

7 Conclusion

We presented SIMD-oriented matrix-layout and folding optimizations for MQOM v2.1 on Arm NEON, AVX2, and AVX-512. Relative to the corresponding v2.1 reference implementations, our optimizations reduce matrix-multiplication cost and peak dynamic memory while improving end-to-end performance across many configurations.

To assess whether these gains carry over to the redesigned scheme, we also applied the matrix-layout optimization to MQOM v3. Our experiments show that the idea remains effective, although the gains depend on the configuration because v3 already incorporates related implementation changes. Further improvements will likely require optimizing the

symmetric primitives, especially Rijndael-256-256 at Categories III and V.

Acknowledgments

The first and second authors gratefully acknowledge the Cryptography Research Center of the Technology Institute of Innovation for partial financial support during the development of this work. We would like to express our sincere gratitude to Michael Scott and Roberto Cabral for their valuable contributions to this paper. Michael provided insightful comments on earlier versions of the paper, especially regarding the side-channel discussion and the need for the row-wise variant. Roberto provided valuable assistance with the AVX2 and AVX-512 experiments. We are sincerely grateful to our reviewers and shepherd; their insightful feedback and attention to detail greatly enhanced the refinement of our manuscript.

References

- [ABB⁺24] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibault Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovahery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. SDitH — Syndrome Decoding in the Head. Technical report, National Institute of Standards and Technology, 2024. Available at <https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures>.
- [AYW⁺21] Monjur Alam, Baki Yilmaz, Frank Werner, Niels Samwel, Alenka Zajic, Daniel Genkin, Yuval Yarom, and Milos Prvulovic. Nonce@once: A single-trace EM side channel attack on several constant-time elliptic curve implementations in mobile platforms. In *Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 507–522. IEEE, 2021.
- [BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security 2003)*, Washington, D.C., 2003. USENIX Association.
- [BBdSG⁺23] Carsten Baum, Lennart Braun, Cyprien Delpéch de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In *Advances in Cryptology – CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part V*, page 581–615, Berlin, Heidelberg, 2023. Springer-Verlag.
- [BBFR25] Ryad Benadjila, Charles Bouillaguet, Thibault Feneuil, and Matthieu Rivain. MQOM: MQ on my Mind Algorithm Specifications and Supporting Documentation (Version 2.1). Technical report, 2025.
- [BBFR26] Ryad Benadjila, Charles Bouillaguet, Thibault Feneuil, and Matthieu Rivain. Mqom: Mq on my mind – algorithm specifications and supporting documentation (version 3.0). Technical report, NIST Post-Quantum Cryptography Additional Digital Signatures, Round 3, August 2026. Version 3.0, August 31, 2026.
- [BF26] Ryad Benadjila and Thibault Feneuil. Breaking the myth of MPCitH inefficiency: Optimizing MQOM for embedded platforms. TCHES, 2026.

- [BMSV22] Emanuele Bellini, Rusydi H. Makarim, Carlo Sanna, and Javier Verbel. An Estimator for the Hardness of the MQ Problem. In *Progress in Cryptology - AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18–20, 2022, Proceedings*, page 323–347, Berlin, Heidelberg, 2022. Springer-Verlag.
- [BZG⁺26] Ryan Babbush, Adam Zalcman, Craig Gidney, Michael Broughton, Tanuj Khattar, Hartmut Neven, Thiago Bergamaschi, Justin Drake, and Dan Boneh. Securing elliptic curve cryptocurrencies against quantum vulnerabilities: Resource estimates and mitigations, 2026.
- [FHK⁺20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Technical report, October 2020. Specification v1.2.
- [FR25] Thibault Feneuil and Matthieu Rivain. Threshold Computation in the Head: Improved Framework for Post-Quantum Signatures and Zero-Knowledge Arguments. *Journal of Cryptology*, 38(3), June 2025.
- [FR26] Thibault Feneuil and Matthieu Rivain. On the security of MPC-in-the-head signatures with correlated GGM trees. Cryptology ePrint Archive, Paper 2026/615, 2026.
- [FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO ’86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1987.
- [IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC)*, pages 21–30. ACM, 2007.
- [Nat01] National Institute of Standards and Technology. Advanced encryption standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce, Washington, DC, November 2001.
- [Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard (ML-DSA). Federal Information Processing Standards Publication 204, U.S. Department of Commerce, Washington, DC, 2024.
- [Nat24b] National Institute of Standards and Technology. Stateless hash-based digital signature standard (SLH-DSA). Federal Information Processing Standards Publication 205, U.S. Department of Commerce, Washington, DC, 2024.
- [Nat26] National Institute of Standards and Technology. Round 3 additional signatures, 2026. Updated September 28, 2026.
- [OFRL25] Eric Oliveira, Décio Gazzoni Filho, Felix Rodrigues, and Julio López. Fast implementation of $GF(2^8)$ inversion with applications to the AES S-box using ARM Helium MVE extensions. In *Anais do XXV Simpósio Brasileiro de Cibersegurança*, pages 1051–1058, Porto Alegre, RS, Brasil, 2025. SBC.
- [Sho94] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 124–134. IEEE Computer Society Press, 1994.