Nibbling MAYO: Optimized Implementations for AVX2 and Cortex-M4

Ward Beullens[1], Fabio Campos[2], Sofía Celi[3], Basil Hess[1] and Matthias J. Kannwischer[4]

[1] IBM Research Europe, Zurich, Switzerland
[2] RheinMain University of Applied Sciences, Wiesbaden, Germany
[3] Brave Software
[4] Quantum Safe Migration Center, Chelpis Quantum Tech, Taipei, Taiwan[†]
contact@pqmayo.org

Abstract. MAYO is a popular high-calorie condiment as well as an auspicious candidate in the ongoing NIST competition for additional post-quantum signature schemes achieving competitive signature and public key sizes. In this work, we present high-speed implementations of MAYO using the AVX2 and Armv7E-M instruction sets targeting recent x86 platforms and the Arm Cortex-M4. Moreover, the main contribution of our work is showing that MAYO can be even faster when switching from a bitsliced representation of keys to a nibble-sliced representation. While the bitsliced representation was primarily motivated by faster arithmetic on microcontrollers, we show that it is not necessary for achieving high performance on Cortex-M4. On Cortex-M4, we instead propose to implement the large matrix multiplications of MAYO using the Method of the Four Russians (M4R), which allows us to achieve better performance than when using the bitsliced approach. This results in up to 21% faster signing. For AVX2, the change in representation allows us to implement the arithmetic much faster using shuffle instructions. Signing takes up to 3.2× fewer cycles and key generation and verification enjoy similar speedups. This shows that MAYO is competitive with lattice-based signature schemes on x86 CPUs, and a factor of 2-6 slower than lattice-based signature schemes on Cortex-M4 (which can still be considered competitive).

Keywords: MAYO · Oil and Vinegar · Arm Cortex-M4 · AVX2 · NIST PQC

1 Introduction

Most public-key cryptographic algorithms that are deployed today are vulnerable to efficient attacks from large-scale quantum computers. Due to this threat, it is important to transition to quantum-safe alternatives. The US National Institute of Standards and Technology (NIST) selected three quantum-safe digital signature algorithms for standardization in 2022 [oST22]: Crystals-Dilithium [LDK+], FALCON [PFH+], and SPHINCS+ [HBD+]. Additionally, NIST is running a process to standardize more quantum-safe signature schemes. One of the most efficient schemes submitted to this process in terms of communication size and speed is MAYO, a multivariate signature scheme.

MAYO [Beu22, BCC+23] is a variant of the *Oil and Vinegar scheme* (OV) [Pat95, KPG99]. The Oil and Vinegar scheme is one of the oldest, and arguably the most studied multivariate digital signature scheme. With small signature sizes, and fast signing and verification, OV has withstood the test of time remarkably well since its invention in 1995.

[†]Part of this work was done while the author was at Academia Sinica.

Its major drawback is its relatively large key sizes. The MAYO variant of the scheme solves the problem of large key sizes while preserving very good computational efficiency and signature size, which makes it a promising candidate in the latest NIST standardization project.

In this work, we focus on implementing the MAYO signature scheme in a high-speed manner. We target both AVX2 and Arm platforms. We follow the specification of the MAYO scheme as given by [BCC+23], but we propose a change to this specification that results in significant implementation speed-ups. The results of our speed-ups can be found in section 6, where we also compare our cycle counts with those of other NIST PQC algorithms.

Contributions. The contribution of our work is fivefold:

- In section 4, we present the first high-speed implementations of the MAYO signature scheme as submitted to the "on-ramp" additional call for quantum-safe signature algorithms by NIST for standardization [NIS22]. We target the AVX2 and the Armv7E-M instruction sets, and present speed records on Intel Skylake, Intel Icelake, and Arm Cortex-M4.
- In section 3, we present a constant-time Gaussian elimination procedure tuned for the MAYO signature scheme with a methodology similar to that of [CKY21], but adapted for non-square matrices.
- In section 5, we propose a change to the current MAYO specification [BCC+23, on 01/06/2023]: While the current version of MAYO uses a bitsliced representation for public keys, private keys, and all outputs of the PRNG, we show that this choice is not ideal. This choice was mainly motivated by platforms that achieve the best performance with bitsliced field arithmetic, such as the Arm Cortex-M4. Platforms for which better arithmetic exists (such as those implementing AVX2 or Arm Neon), suffer with this choice. We instead propose using the nibble-sliced representation[1], which is commonly found in other multivariate cryptosystems such as OV [BCH+23].
- In section 5, we propose to use the Method of the Four Russians (M4R) [ADKF70, AH74] for costly matrix multiplications within MAYO on the Cortex-M4. This essentially trades field multiplications for table look-ups with the latter being much cheaper on embedded platforms. These efficiency gains motivate our usage of M4R. Note that this method is compatible with other multivariate cryptosystems, specially those that use a nibble-sliced representation, such as OV [BCH+23]. However, determining if OV implementations using M4R are superior is not obvious and left to future work. Using M4R, we achieve modest speed-ups of up to 21% over the previous implementations that use the bitsliced representation. However, M4R can only be efficiently implemented if matrices are in nibble-sliced representation. On-the-fly conversion outweighs the gains achieved.
- Using the nibble-sliced representation allows us to implement the $\mathbb{F}_{16}$ arithmetic within MAYO using AVX2 shuffle instructions, which results in much better performance. Fundamentally, this technique is also based on M4R. Using AVX2 shuffle instructions for field multiplication has been common practice in multivariate cryptography for many years [CCC+09, DCP+20, BCH+23]. However, unlike existing approaches, we use both the high and the low nibbles of the lookup table AVX2 register, which doubles the number of multiplications per shuffle instruction. Compared to the bitsliced implementation, the resulting nibble-sliced implementations of MAYO uses up to 3.2× fewer cycles.

[1] The authors of [BCC+23] have agreed to incorporate these changes in the round-2 submission of their specification.

Source code. The source code of the implementations described in this paper is available under an Apache 2.0 license. The reference implementation and the AVX2 implementation are available at https://github.com/PQCMayo/MAYO-C. The Arm Cortex-M4 implementation is available at https://github.com/PQCMayo/MAYO-M4. The bitsliced and nibble-sliced variants are available in separate branches.

Related work. Most prior work [KKS+21, CKY21, Pet13, FG18] on implementations of multivariate signature schemes targets the Rainbow [DCK+21] cryptosystem, since it was a finalist of the NIST Post-Quantum-Cryptography (PQC) standardization process [oST]. However, many of these techniques can be adapted to other OV-based schemes including MAYO. In [Beu22], Beullens provides a preliminary implementation of MAYO. In [BCC+23], the authors present an updated set of parameters and, accordingly, a reference software implementation based on bitsliced arithmetic. In [GMSS23], the authors present the first implementation of MAYO on Arm microcontrollers. They use a modified parameter set to speed up the signing and verification processes, which is very close but not identical to [BCC+23]. We vastly outperform these implementations. It is worth noting that two implementations of MAYO on FPGA were recently proposed [SMA+23, HSMR23] (we include some numbers of the latter in Table 6).

2 Preliminaries

In this section, we recall the MAYO signature scheme (subsection 2.1)[2] and the Method of the Four Russians (subsection 2.2).

Notation. If X is a finite set, we write $x \xleftarrow{\$} X$ to denote that x is assigned a value chosen from X uniformly at random. If A is an algorithm, we write $x \leftarrow A(y)$ to denote that x is assigned the output of running A on input y. If k is an integer, we denote by $[k]$ the set $\{0, \ldots, k-1\}$. We denote by $\{x_i\}_{i\in[k]}$ a sequence of objects $x_0, \ldots, x_{k-1}$ indexed by elements of $[k]$. We denote the base-2 logarithm by log, and we denote binomial coefficients by $\binom{n}{k}$, i.e., $\binom{n}{k} = n!/k!(n-k)!$. We use the standard Landau notation $O(\cdot)$ for asymptotics.

We denote by $\mathbb{F}_q$ a finite field with q elements and by $\mathbb{F}_q^{m\times n}$ the set of (zero-indexed) matrices over $\mathbb{F}_q$ with m rows and n columns. We denote by $\mathbf{I}_a \in {\mathbb{F}_q}^{a\times a}$ the identity matrix of size a-by-a. If $\mathbf{A} \in {\mathbb{F}_q}^{m\times n}$ and $\mathbf{b} \in {\mathbb{F}_q}^{m}$, we denote by $\mathbf{A}[i,j]$ the entry in the i-th row and the j-th column of $\mathbf{A}$, by $\mathbf{A}[:,i] \in {\mathbb{F}_q}^{m}$ the i-th column of $\mathbf{A}$, and by $\mathbf{A}[i,:] \in {\mathbb{F}_q}^{n}$ the i-th row of $\mathbf{A}$. We denote by $(\mathbf{A}\,\mathbf{b}) \in {\mathbb{F}_q}^{m\times(n+1)}$ the matrix whose first n columns are the columns of $\mathbf{A}$, and whose last column is $\mathbf{b}$. We say a matrix $\mathbf{A} \in {\mathbb{F}_q}^{n\times n}$ is upper triangular if $\mathbf{A}[i,j] = 0$ for all $0 \le j < i < n$.

2.1 MAYO

Both an *Oil and Vinegar* [KPG99, Pat97] and a MAYO public key represents a multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ consisting of m homogeneous quadratic polynomials in n variables over a small finite field $\mathbb{F}_q$. The secret key represents a linear subspace $O \subset \mathbb{F}_q^n$ of dimension o, on which $\mathcal{P}$ vanishes, i.e. $\mathcal{P}(\mathbf{o}) = 0$ for all vectors $\mathbf{o} \in O$. In the case of *Oil and Vinegar*, $o = m$, and $\mathcal{P}$ is used directly to verify if a signature $\mathbf{s} \in \mathbb{F}_q^n$ is valid for a message m given that public key $\mathcal{P}$: the signature is valid if $\mathcal{P}(\mathbf{s}) = \mathcal{H}(m)$, where $\mathcal{H}$ is a salted hash function that outputs elements in $\mathbb{F}_q^m$. Knowledge of the secret space O allows the signer to sample such signatures by solving a system of m linear equations.

[2] For an in-depth explanation, see [BCC+23, Chapter 1 & 2]

MAYO is a variant of the *Oil and Vinegar* scheme, where $\mathcal{P}$ has the same structure with the exception that the dimension of the space O on which $\mathcal{P}$ evaluates to zero is "too small", i.e., $\dim(O) = o$, with o less than m. Reducing the dimension of O drastically shrinks the key sizes. However, it also means that the OV signing algorithm does not work anymore. To solve this problem, $\mathcal{P}$ is not used directly in the signature and verification procedure. Instead, the verifier "whips up" $\mathcal{P}$ into a k-fold larger map $\mathcal{P}^\star : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$, with m polynomials in k sets of n variables (k is a parameter of the scheme). Concretely, $\mathcal{P}^\star$ is defined as:

$$\mathcal{P}^\star(\mathbf{x}_1, \ldots, \mathbf{x}_k) := \sum_{i=1}^{k} \mathbf{E}_{ii}\mathcal{P}(\mathbf{x}_i) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{x}_i, \mathbf{x}_j)\,,$$

where $\mathcal{P}'(\mathbf{x}, \mathbf{y}) := \mathcal{P}(\mathbf{x} + \mathbf{y}) - \mathcal{P}(\mathbf{x}) - \mathcal{P}(\mathbf{y})$, and where for all $i \in \{1, \ldots, k\}$ and all $j \in \{i+1, \ldots, k\}$ the matrix $\mathbf{E}_{ij} \in \mathbb{F}_q^{m \times m}$ is fixed and public. These matrices are chosen such that, under the correspondence between vectors in $\mathbb{F}_q^m$ and polynomials in $\mathbb{F}_q[X]$ of degree at most m, multiplication by $\mathbf{E}_{ij}$ corresponds to multiplication by powers of X modulo an irreducible polynomial $f(X) \in \mathbb{F}_q[X]$ of degree m. A MAYO signature $\mathbf{S} = (\mathbf{s}_1, \ldots, \mathbf{s}_k) \in \mathbb{F}_q^{nk}$ is considered valid if $\mathcal{P}^\star(\mathbf{s}_1, \ldots, \mathbf{s}_k) = \mathcal{H}(m)$.

To compute $\mathcal{P}^\star(\mathbf{S})$, the verifier (as seen in Algorithm 3) first computes $\mathcal{P}(\mathbf{s}_i)$ and $\mathcal{P}'(\mathbf{s}_i, \mathbf{s}_j)$ for all $i \in \{1, \ldots, k\}$ and all $j \in \{i+1, \ldots, k\}$, and then combines said variables to obtain $\mathcal{P}^\star(\mathbf{s})$. Since the $\mathbf{E}_{ij}$ matrices act as multiplication by powers of $X \pmod{f(X)}$, the verifier can multiply the polynomials corresponding to $\mathcal{P}(\mathbf{s}_i)$ and $\mathcal{P}'(\mathbf{s}_i, \mathbf{s}_j)$ with the appropriate powers of X and perform a single reduction modulo $f(X)$. Computing the evaluations of $\mathcal{P}$ and $\mathcal{P}'$ is computationally more demanding than combining the results.

Similarly, to sign a message (as seen in Algorithm 2), the signer has to partially evaluate $\mathcal{P}$ and $\mathcal{P}'$ on k vectors $(\mathbf{v}_1, \ldots, \mathbf{v}_k) \in \mathbb{F}_q^{n-o}$, and combine the results to calculate the coefficients of a system of linear equations, $\mathbf{A}\mathbf{x} = \mathbf{y}$, whose solution will determine a signature. The most computationally demanding steps of signing are the partial evaluations of $\mathcal{P}$ and $\mathcal{P}'$ and the Gaussian elimination used to solve the linear system. Hence, these should be the main focus of optimization efforts. In contrast, the task of combining the partial evaluations into $\mathbf{A}$ and $\mathbf{y}$, and the task of obtaining a signature $\mathbf{s}$ from a solution $\mathbf{x}$ to the system $\mathbf{A}\mathbf{x} = \mathbf{y}$ accounts for only a small fraction of the signing time, and, therefore, does not need careful optimization.

Polynomial evaluation as matrix multiplication. The $\binom{n+1}{2}$ coefficients of each of the m polynomials $(p_1, \ldots, p_m)$ in the MAYO public key $\mathcal{P}$ are arranged in the upper-diagonal part of n-by-n matrices $\mathbf{P}_k$ such that

$$p_k(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{P}_k\mathbf{x}$$

for all $1 \leq k \leq m$. Moreover, we have

$$p'_k(\mathbf{x}, \mathbf{y}) := p_i(\mathbf{x} + \mathbf{y}) - p_i(\mathbf{x}) - p_i(\mathbf{y}) = \mathbf{x}^\mathsf{T}\mathbf{P}_k\mathbf{y} + \mathbf{y}^\mathsf{T}\mathbf{P}_k\mathbf{x}\,.$$

The matrices $\mathbf{P}_k$ are split in 3 parts as follows

$$\mathbf{P}^{(k)} = \begin{pmatrix} \mathbf{P}_k^{(1)} & \mathbf{P}_k^{(2)} \\ 0 & \mathbf{P}_k^{(3)} \end{pmatrix},$$

where $\mathbf{P}_k^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_k^{(3)} \in \mathbb{F}_q^{o\times o}$ are upper-diagonal, and $\mathbf{P}_k^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$. The matrices $\mathbf{P}_k^{(1)}$ and $\mathbf{P}_k^{(2)}$ are expanded from a short seed using an AES-based expansion function, while $\mathbf{P}_k^{(3)}$ is stored as part of the public key.

To compute $\mathcal{P}(\mathbf{s}_i)$ and $\mathcal{P}'(\mathbf{s}_i, \mathbf{s}_j)$ for all $1 \leq i < j \leq k$, it suffices to compute $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$ for all $k \in [m]$, where $\mathbf{S} \in {\mathbb{F}_q}^{n \times k}$ is the matrix whose columns are $\mathbf{s}_1, \ldots, \mathbf{s}_k$. The value of $\mathcal{P}(\mathbf{s}_i)_k$ can be found on the diagonal of $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$ and $\mathcal{P}'(\mathbf{s}_i, \mathbf{s}_j)_k$ is the sum of the entries at locations (i, j) and (j, i) in the matrix $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$.

Matrix-matrix multiplications (with the left matrix possibly being upper-diagonal) are used extensively as part of the signing and verification algorithms of MAYO, which means they should be implemented and optimized carefully.

In Algorithm 1 (KeyGen), Algorithm 2 (Sign), and Algorithm 3 (Verify), we give simplified pseudocode for the MAYO signature scheme, but for a detailed specification we refer to [BCC+23]. In particular, we refer to the full specification for the Compute_y and Compute_A functions, that respectively compute the right-hand side and the left-hand side of the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{y}$. Implementing these functions is relatively straightforward and cheap and was not the focus of the optimization effort of this paper.

Algorithm 1 KeyGen ()

Output: A key pair $(\mathsf{pk}, \mathsf{sk})$

1: //Derive $\mathbf{O}$ and the $\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}$ from random $\mathsf{seed}_\mathsf{sk}$.
2: $\mathsf{seed}_\mathsf{sk} \xleftarrow{\$} \{0,1\}^{\lambda+64}$
3: $(\mathsf{seed}_\mathsf{pk}, \mathbf{O}) \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_\mathsf{sk})$ // $\mathbf{O} \in {\mathbb{F}_q}^{(n-o)\times o}$
4: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_\mathsf{pk})$ // $\mathbf{P}_i^{(1)} \in {\mathbb{F}_q}^{(n-o)\times(n-o)}, \mathbf{P}_i^{(2)} \in {\mathbb{F}_q}^{(n-o)\times o}$
5: //Compute $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$.
6: **for** i from 0 to $m-1$ **do**
7: $\quad \mathbf{P}_i^{(3)} \leftarrow \mathsf{Upper}(-\mathbf{O}^\mathsf{T}(\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{P}_i^{(2)}))$
8: **return** $(\mathsf{pk} = (\mathsf{seed}_\mathsf{pk}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]}), \mathsf{sk} = \mathsf{seed}_\mathsf{sk})$.

Note that MAYO sets the size of the finite field to be 16: $\mathbb{F}_{16}$. It also provides 4 parameter sets: MAYO_1, MAYO_2, MAYO_3 and MAYO_5. The first two parameters are for NIST security level 1, the third for NIST security level 3, and the fourth for NIST security level 5.

2.2 Method of the Four Russians

The Method of the Four Russians (M4R) was first presented by Arlazarov, Dinic, Kronrod, and Faradzev [ADKF70] and received its name in [AH74, Chapter 6]. It was originally presented for multiplying boolean matrices, but it can be straightforwardly extended for matrix multiplication over small fields, and in particular over $\mathbb{F}_{16}$.

For matrix multiplication, the algorithm works as follows: given a small integer t, to compute the product of a $(n \times m)$ matrix $\mathbf{A}$ and a $(m \times k)$ matrix $\mathbf{B}$, one divides $\mathbf{A}$ into m/t vertical stripes $\mathbf{A}_i$, and $\mathbf{B}$ into m/t horizontal stripes $\mathbf{B}_i$, which allows the product $\mathbf{AB}$ to be computed as $\sum_{i=0}^{k} \mathbf{A}_i\mathbf{B}_i$. Multiplication, then, works as follows:

- For each stripe, compute all linear combinations of the rows of $\mathbf{B}_i$ as a look-up table T to store $16^t \cdot k$ field elements.
- Use each row in $\mathbf{A}_i$ as an index to look up the corresponding row from T and accumulate the product.

If $t = 2$, we can illustrate the method with the following example, given the following matrix product:

Algorithm 2 Sign $(\mathsf{seed}_{\mathsf{sk}}, M)$

Input: Secret key $\mathsf{seed}_{\mathsf{sk}}$
Input: Message M
Output: Signature $(\mathbf{S}, \mathsf{salt})$

1: //Rederive $\mathbf{O}$ and $\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}$ from $\mathsf{seed}_{\mathsf{sk}}$.
2: $(\mathsf{seed}_{\mathsf{pk}}, \mathbf{O}) \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_{\mathsf{sk}})$
3: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_{\mathsf{pk}})$
4: //Hash salted message.
5: $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{\lambda+64}$
6: $\mathbf{t} \leftarrow \mathsf{SHAKE256}(M\|\mathsf{salt})$ // $\mathbf{t} \in {\mathbb{F}_q}^m$
7: $\mathbf{V} \xleftarrow{\$} {\mathbb{F}_q}^{k\times(n-o)}$
8: **for** i from 1 to m **do**
9: $\quad \mathbf{L}_i \leftarrow (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}) \cdot \mathbf{O} + \mathbf{P}_i^{(2)}$ // $\mathbf{L}_i \in {\mathbb{F}_q}^{(n-o)\times o}$
10: $\quad \mathbf{M}_i \leftarrow \mathbf{V} \cdot \mathbf{L}_i$ // $\mathbf{M}_i \in {\mathbb{F}_q}^{k\times o}$
11: $\quad \mathbf{Y}_i \leftarrow \mathbf{V} \cdot \mathbf{P}_i^{(1)} \cdot \mathbf{V}^{\mathsf{T}}$ // $\mathbf{Y}_i \in {\mathbb{F}_q}^{k\times k}$
12: //Build linear system $\mathbf{Ax} = \mathbf{y}$.
13: $\mathbf{A} \leftarrow \mathsf{Compute\_A}(\{\mathbf{M}_i\}_{i\in[m]})$
14: $\mathbf{y} \leftarrow \mathbf{t} + \mathsf{Compute\_y}(\{\mathbf{Y}_i\}_{i\in[m]})$
15:
16: //Try to sample a random solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{y}$.
17: $\mathbf{x} \leftarrow \mathsf{SampleSolution}(\mathbf{A}, \mathbf{y})$ // $\mathbf{x} \in {\mathbb{F}_q}^{ko} \cup \{\perp\}$
18: **if** $\mathbf{x} = \perp$ **then** // Retry if there are no solutions
19: $\quad$ **go to** 7
20: //Output the signature.
21: $\mathbf{X} \leftarrow \mathsf{Matrixify}(\mathbf{x})$ // $\mathbf{X} \in {\mathbb{F}_q}^{k\times o}$, s.t. $\mathbf{x}$ is concatenation of rows of $\mathbf{X}$
22: $\mathbf{S} \leftarrow (\mathbf{V} + (\mathbf{OX})^{\mathsf{T}}, \mathbf{X})$ // $\mathbf{S} \in {\mathbb{F}_q}^{k\times n}$
23: **return** $(\mathbf{S}, \mathsf{salt})$.

Algorithm 3 Verify $(\mathsf{pk}, M, \mathsf{Sig})$

Input: Public key $\mathsf{pk} = (\mathsf{seed}_{\mathsf{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
Input: Message M
Input: Signature $\mathsf{Sig} = (\mathbf{S}, \mathsf{salt})$
Output: An boolean to indicate if the signature is valid.

1: //Derive $\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$.
2: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_{\mathsf{pk}})$
3: //Hash salted message.
4: $\mathbf{t} \leftarrow \mathsf{SHAKE256}(M\|\mathsf{salt})$ // $\mathbf{t} \in {\mathbb{F}_q}^m$
5: //Compute $\mathcal{P}^*(\mathbf{s})$.
6: **for** i from 1 to m **do**
7: $\quad \mathbf{Y}_i \leftarrow \mathbf{S} \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{pmatrix} \mathbf{S}^{\mathsf{T}}$
8: $\mathbf{y} \leftarrow \mathsf{Compute\_y}(\{\mathbf{Y}_i\}_{i\in[m]})$ // $\mathbf{y} = \mathcal{P}^*(\mathbf{s})$
9: **return** $\mathbf{y} == \mathbf{t}$ // Accept signature if $\mathbf{y} = \mathbf{t}$.

$$\mathbf{AB} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n-1,0} & a_{n-1,1} & a_{n-1,2} & a_{n-1,3} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \\ b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}$$

To compute this product using M4R (with $t = 2$), we split the matrices into stripes:

$$\mathbf{AB} = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \\ \vdots & \vdots \\ a_{n-1,0} & a_{n-1,1} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \\ \vdots & \vdots \\ a_{n-1,2} & a_{n-1,3} \end{bmatrix} \begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}$$

For each stripe of $\mathbf{B}$, we now compute a 256-entry look-up table T with each entry containing $k = 2$ field elements. Then, we go through the stripes of $\mathbf{A}$ and use $(a_{i,j}, a_{i,j+1})$ as index to the look-up table T.

Using M4R in cryptography. As the method uses look-up tables, one has to be careful to not leak secret data through the addresses used for look-ups. It is, hence, essential to either only use this trick when the matrix used for indexing is public or make use of constant-time table look-ups. Luckily, the former is the case for all major matrix multiplications in MAYO and the latter can be used in AVX2.

3 System-solving using Gaussian elimination.

In this section, we describe how the system of linear equations is solved during signing in our MAYO implementations, which is independent of the proposed change in representation (independent of both bitsliced and nibble-based representations taking into account the considerations presented in the following paragraphs).

In the MAYO signing algorithm, the signer samples a uniformly random solution (if it exists) to a system of linear equations $\mathbf{Ax} = \mathbf{y}$ for a rectangular matrix $\mathbf{A} \in {\mathbb{F}_q}^{m \times ko}$ and $\mathbf{y} \in {\mathbb{F}_q}^m$. Our implementations use constant-time Gaussian elimination to solve this problem. To randomize the solving procedure, we sample a random vector $\mathbf{r} \in {\mathbb{F}_q}^{ko}$ and set $\mathbf{y}' = \mathbf{y} + \mathbf{Ar}$. Then, we solve the system $\mathbf{Ax}' = \mathbf{y}'$ for $\mathbf{x}'$ using Gaussian elimination on the augmented matrix $\begin{pmatrix} \mathbf{A} & \mathbf{y}' \end{pmatrix}$, and output $\mathbf{x} = \mathbf{x}' - \mathbf{r}$. Note that $\mathbf{x}$ is a solution because $\mathbf{Ax} = \mathbf{Ax}' - \mathbf{Ar} = \mathbf{y} + \mathbf{Ar} - \mathbf{Ar}$, and one can check that if $\mathbf{r}$ is chosen uniformly at random, then $\mathbf{x}$ is a uniformly random solution to $\mathbf{Ax} = \mathbf{y}$. While it is possible to sample a random solution directly, this method was chosen in [BCC+23] because it is simple to implement in constant time.

This constant-time Gaussian elimination procedure consists of ko iterations: one for each column of $\mathbf{A}$. Initially, we start with $R = 0$ and maintain the invariant that the top R rows of $\mathbf{A}$ form a full-rank matrix in row echelon form with leading ones (i.e. the first non-zero entry of each row is equal to 1 and it is strictly left of the non-zero entries in the rows below it). At iteration i:

- if there are no non-zero entries in column i in rows below R, nothing is done,
- otherwise, we perform elementary row operations on the bottom $m - R$ rows to force a 1 on position $(R + 1, i)$ and zeros on all the entries below it. We set $R := R + 1$.

Care needs to be taken to avoid leaking the value R throughout the Gaussian elimination process, e.g., we cannot use R as an index to directly read and write from $\mathbf{A}$. We follow a methodology similar to that of [CKY21], adapted for non-square matrices. To create row

$R+1$, we move through r from 1 to m, and, conditionally (but in constant time), add row r of $\mathbf{A}$ into a buffer row $\mathbf{b}$, if $r > R$ and if the pivot in the buffer is zero. Then, we multiply the buffered row by $\mathbf{b}_i^{q-2}$ (which is equal to the inverse of $\mathbf{b}_i$ if $\mathbf{b}_i \neq 0$). This procedure ensures that either $\mathbf{b} = 0$, if there was no pivot available in column i; or, otherwise, $\mathbf{b}$ is a linear combination of rows below R with a 1 in location i. Subsequently, we scan through the rows of $\mathbf{A}$ and conditionally write $\mathbf{b}$ to row r if $r = R+1$ and if $\mathbf{b}_i$ is nonzero. We conditionally add $-a_{r,i}\mathbf{b}$ to row r of $\mathbf{A}$ if $r > R+1$. This constant-time version of Gaussian elimination is slower than the usual variable-time version, but both versions have an asymptotic complexity of $O(m^2ko)$.

Platform-specific considerations. Our AVX2 implementation stores rows of the augmented matrix in the low nibbles of 256-bit vectors. This allows for efficient elementary row operations using `vpxor` and `vpshufb` instructions because the `vpshufb` instruction does table lookups using the low nibbles as indices. On the Cortex-M4 platform, we keep the rows of the augmented matrix in bitsliced representation throughout the Gaussian elimination. We bitslice them in the beginning and un-bitslice at the end. [CKY21] states that bitslicing is undesirable as individual elements have to be accessed as pivots requiring to un-do the bitslicing. We work around this by extracting only the pivot elements from the bitsliced representation which we achieve in 14 instructions (excluding memory operations). While this is costly, it is still a performance improvement over previous implementations.

4 Bitsliced MAYO implementation

In this section, we present our implementation using the bitsliced representation for MAYO keys and PRNG output. This implementation is compatible with the MAYO round-1 specification as submitted to NIST PQC process. We present both our AVX2 and Arm Cortex-M4 implementations with this representation. Details on the AVX2 instruction set can be found in [int] and for the Cortex-M4 instruction set in [arm]. Both the bitsliced and the improved nibble-sliced implementation don't perform any secret dependent branching and table lookups: we validate that property in the AVX2 implementations using Valgrind and the ctgrind [Lan10] method as part of our test harness.

4.1 AVX2

Bitsliced arithmetic in AVX2. In our bitsliced AVX2 implementation, we fit 64 elements of $\mathbb{F}_{16}$ in an AVX2 register in a bit-interleaved fashion: the least significant bits of the 64 elements go to the first 64 bits of the AVX registers, the 2nd bits of the elements go to bits 65 to 128 in the AVX2 register, and so on. Adding two vectors of 64 field elements, then simply corresponds to XORing the corresponding AVX2 registers. Moreover, we can multiply the 64 field elements with a scalar $b \in \mathbb{F}_{16}$ by using 17 AVX2 instructions (8 `vpand`, 4 `vpcmpeqq`, 3 `vpxor`, 2 `vpshufd`, 1 `vpbroadcastb`, 1 `vpermpd`, and 1 `vpshufb`), as shown in Figure 1.

Note that because of bitslicing, each 64-bit block of the output is an $\mathbb{F}_2$-linear combination of the four 64-bit blocks in the input register, where the linear combination depends on the scalar b. For example, if $b = x+1$, then multiplication by b maps $a_0+a_1x+a_2x^2+a_3x^3$ to $(a_0+a_3)+(a_0+a_1+a_3)x+(a_1+a_2)x^2+(a_2+a_3)x^3$, so, if the 64-bit blocks in the input register are A_0, A_1, A_2, A_3, respectively, then the first 64-bit block of the output should be $A_0 \oplus A_3$, and the next 64-bit block should be $A_0 \oplus A_1 \oplus A_3$, etc.

Our method for doing the scalar multiplication on 64 bitsliced field element first uses two `vpshufd` instructions and one `vpermpd` instruction, to shuffle around the 64-bit blocks of the input. This gives us four 256-bit vectors, such that each 64-bit block of the input appears at locations 1-64, 65-128, 129-192, and 193-256 in one of the four vectors. Therefore,

```
static
inline void bitsliced_64_vec_mul(const __m256i *in, unsigned char b, __m256i *out){
    // prepare constants
    const __m256i lut_b = _mm256_setr_epi8(
        0x00, 0x13, 0x26, 0x35, 0x4c, 0x5f, 0x6a, 0x79,
        0x98, 0x8b, 0xbe, 0xad, 0xd4, 0xc7, 0xf2, 0xe1,
        0x00, 0x13, 0x26, 0x35, 0x4c, 0x5f, 0x6a, 0x79,
        0x98, 0x8b, 0xbe, 0xad, 0xd4, 0xc7, 0xf2, 0xe1);
    const __m256i mask1 = _mm256_set_epi64x(16,  16, 16, 1  );
    const __m256i mask2 = _mm256_set_epi64x(32,  8,  32, 128);
    const __m256i mask3 = _mm256_set_epi64x(64,  64, 4,  64 );
    const __m256i mask4 = _mm256_set_epi64x(128, 32, 8,  32 );

    // permute quadwords
    __m256i in_1234 = *in;
    __m256i in_3412 = _mm256_permute4x64_epi64(in_1234, 0b01001110);
    __m256i in_2143 = _mm256_shuffle_epi32(in_1234, 0b01001110);
    __m256i in_4321 = _mm256_shuffle_epi32(in_3412, 0b01001110);

    // mask and combine
    __m256i lookup = _mm256_shuffle_epi8(lut_b, _mm256_set1_epi8(b));
    *out = in_1234 & _mm256_cmpeq_epi64(lookup & mask1, mask1);
         ^ in_2143 & _mm256_cmpeq_epi64(lookup & mask2, mask2);
         ^ in_3412 & _mm256_cmpeq_epi64(lookup & mask3, mask3);
         ^ in_4321 & _mm256_cmpeq_epi64(lookup & mask4, mask4);
}
```

Figure 1: `C` code with Intel intrinsics for multiplying 64 bitsliced field elements by the element $\mathtt{b} \in \mathbb{F}_{16}$.

the result of the scalar multiplication can be formed by masking out 64-bit blocks of these 4 vectors and XORing the results together (4 `vpand` and 4 `vpxor` instructions). The masks are created from b using one `vpbroadcastb`, one `vpshufb`, 4 `vpands`, 4 `vpcmpeqq` instructions, and five pre-loaded 256-bit vectors.

For the MAYO_1 and MAYO_2 parameter sets, we require scalar multiplication of vectors of length 64, which perfectly fits into one AVX2 register as described above. The MAYO_3 and MAYO_5 parameter sets require scalar multiplication of vectors of length 96 and 128, respectively. We use an analogous strategy: for MAYO_3 we use three 128-bit SSE2 vectors to store the vector, and for MAYO_5 we use two 256-bit AVX2 vectors.

Matrix multiplications. The matrix multiplications that need to be performed inside KeyGen, Sign, and Verify comes in batches of size $m \in \{64, 96, 128\}$. For example, during key generation, the m matrices $\mathbf{P}_1^{(1)}, \ldots, \mathbf{P}_m^{(1)}$ are all multiplied by $\mathbf{O}$ from the right (see line 7 of Algorithm 1). We represent and sample a batch of m matrices $\mathbf{M}^{(1)}, \ldots, \mathbf{M}^{(m)} \in {\mathbb{F}_q}^{n \times m}$ in a doubly interleaved format, such that for each location (i, j), all m field elements $\mathbf{M}_{i,j}^{(1)}, \ldots, \mathbf{M}_{i,j}^{(m)}$ sit contiguously in memory using the bitsliced representation previously described. Multiplying a batch of m matrices by a single matrix can then be done in parallel using vector additions and vector scalar multiplications.

4.2 Arm Cortex-M4

Matrix multiplications. We borrow the bitsliced arithmetic from [CKY21], which is straightforwardly extended to all matrix multiplications required in MAYO. Due to the bitsliced representation of the public key and the sampled matrices, no additional bitslicing operation is required: this dramatically improves performance.

Input: Accumulators $a_1, \ldots, a_{15}$
Output: $a_1 + a_2 \cdot x + \cdots + a_{15} \cdot (x^3 + x^2 + x + 1)$

Algorithm 4 Method 1

1: $r = a_1 + a_2 \cdot (x)$
2: $r \mathrel{+}= a_3 \cdot (x + 1)$
3: $r \mathrel{+}= a_4 \cdot (x^2)$
4: $r \mathrel{+}= a_5 \cdot (x^2 + 1)$
5: $r \mathrel{+}= a_6 \cdot (x^2 + x)$
6: $r \mathrel{+}= a_7 \cdot (x^2 + x + 1)$
7: $r \mathrel{+}= a_8 \cdot (x^3)$
8: $r \mathrel{+}= a_9 \cdot (x^3 + 1)$
9: $r \mathrel{+}= a_{10} \cdot (x^3 + x)$
10: $r \mathrel{+}= a_{11} \cdot (x^3 + x + 1)$
11: $r \mathrel{+}= a_{12} \cdot (x^3 + x^2)$
12: $r \mathrel{+}= a_{13} \cdot (x^3 + x^2 + 1)$
13: $r \mathrel{+}= a_{14} \cdot (x^3 + x^2 + x)$
14: $r \mathrel{+}= a_{15} \cdot (x^3 + x^2 + x + 1)$
15: **return** r

Algorithm 5 Method 2

1: $a_{12} \mathrel{+}= a_{15}; a_3 \mathrel{+}= a_{15}$
2: $a_8 \mathrel{+}= a_{14}; a_6 \mathrel{+}= a_{14}$
3: $a_{10} \mathrel{+}= a_{13}; a_7 \mathrel{+}= a_{13}$
4: $a_8 \mathrel{+}= a_{12}; a_4 \mathrel{+}= a_{12}$
5: $a_9 \mathrel{+}= a_{11}; a_2 \mathrel{+}= a_{11}$
6: $a_8 \mathrel{+}= a_{10}; a_2 \mathrel{+}= a_{10}$
7: $a_8 \mathrel{+}= a_9; a_1 \mathrel{+}= a_9$
8: $a_4 \mathrel{+}= a_7; a_3 \mathrel{+}= a_7$
9: $a_4 \mathrel{+}= a_6; a_2 \mathrel{+}= a_6$
10: $a_4 \mathrel{+}= a_5; a_1 \mathrel{+}= a_5$
11: $a_2 \mathrel{+}= a_3; a_1 \mathrel{+}= a_3$
12: $r = a_4 + a_8 \cdot x$
13: $r = a_2 + r \cdot x$
14: $r = a_1 + r \cdot x$
15: **return** r

Algorithm 6 Method 3

1: $a_{10} \mathrel{+}= a_5 \cdot x^{-1}$
2: $a_{12} \mathrel{+}= a_{11} \cdot x$
3: $a_7 \mathrel{+}= a_{10} \cdot x^{-1}$
4: $a_6 \mathrel{+}= a_{12} \cdot x$
5: $a_{14} \mathrel{+}= a_7 \cdot x^{-1}$
6: $a_3 \mathrel{+}= a_6 \cdot x$
7: $a_{15} \mathrel{+}= a_{14} \cdot x^{-1}$
8: $a_8 \mathrel{+}= a_3 \cdot x$
9: $a_{13} \mathrel{+}= a_{15} \cdot x^{-1}$
10: $a_4 \mathrel{+}= a_8 \cdot x$
11: $a_9 \mathrel{+}= a_{13} \cdot x^{-1}$
12: $a_2 \mathrel{+}= a_4 \cdot x$
13: $a_1 \mathrel{+}= a_9 \cdot x^{-1}$
14: $a_1 \mathrel{+}= a_2 \cdot x$
15: **return** a_1

Figure 2: Different methods for obtaining the final accumulated result in the evaluation of multivariate polynomials after using the trick from [CKY21].

Verification. We make use of the method for computing $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$ presented in [CKY21]. However, as $\mathbf{S}$ is a matrix for MAYO (rather than a vector as in Rainbow and OV), we cannot efficiently compute $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$ in a single pass. We instead, first compute both $\mathbf{P}_k\mathbf{S}$ and $\mathbf{S}^\mathsf{T}\mathbf{P}_k\mathbf{S}$ with the method of using 16 (for $\mathbb{F}_{16}$) accumulators to minimize the number of field multiplications. There is one notable difference: when computing the public map in a single pass, one can omit a large portion of the computation for each variable that is zero. Implementations of OV [BCH+23] and Rainbow [CKY21] explicitly check for zero variables in the outer loop and skip ahead. This also allows working with only 15 accumulators instead of 16. However, said trick does not help when computing the two products separately and results in a slowdown. We, hence, do not check for zero variables and use 16 accumulators instead.

While the remaining multiplications to compute the final result in OV and Rainbow are negligible, the number of remaining multiplications in MAYO is significantly higher making up a large portion of the total runtime. It is, hence, important to consider the best strategy for performing those multiplications. We consider 3 different methods for performing the multiplications, which we describe in Figure 2. [CKY21] and [BCH+23] use Method 1 as the number of multiplications is negligible. This method uses 14 multiply-accumulate operations. Method 2 is described in [CKY21], but not implemented. It uses the minimum number of multiplications (3 multiply-accumulate operations, and 22 addition operations). Method 3 uses 14 multiply-accumulate operations but only uses multiplications by x and x^{-1} which can be implemented much more efficiently than general-purpose multiplications. Choosing between Method 2 and Method 3 depends on the cost ratio between multiplications and additions. For the bitsliced variant, Method 3 is faster as multiplication by x or x^{-1} can be implemented in just 5 `eor` instructions (operating on 32 field elements in parallel).

5 Improving the MAYO implementation

In this section, we describe the proposed change of the representation of MAYO's keys and PRNG output to a nibble-sliced representation. This implementation of this method is not compatible with round-1 MAYO as submitted to the NIST PQC process[3]. We also

[3]The authors of [BCC+23] have agreed to incorporate these changes in the round-2 submission of their specification.

examine how we use M4R, and propose the changes to both the AVX2 and Arm Cortex-M4 implementations.

Proposed specification change. Our proposed change concerns the representation and sampling of the matrices $\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}, \mathbf{P}_i^{(3)}$, and $\mathbf{L}_i$. In the round-1 submission, MAYO uses a bitsliced representation for the batch of matrices $\mathbf{A}_0, \ldots, \mathbf{A}_{m-1}$. The representation encodes elements of the vectors $(\mathbf{A}_0[i,j], \ldots, \mathbf{A}_{m-1}[i,j])$ in a bitsliced fashion meaning the least significant bits occupy the first m bits. The bitsliced vectors are then stored in a column-major form. We propose to discard the bitslicing and, instead, store two field elements packed into one byte with the first element in the least significant four bits. The order of the element batches remains the same. This corresponds to the common column-major Macaulay matrix representation in lexicographic order. Note that this change modifies both the sampling process and the public key format. It also modifies the format of the expanded secret key.

5.1 AVX2

M4R in MAYO on AVX2. We implement M4R on AVX2 to perform the various matrix multiplications performed inside MAYO. We take advantage of `vpshufb` instructions instead of traditional table lookups to speed up our implementation. A single `vpshufb` instruction corresponds to 32 lookups in a table with 16 bytes. Since the size of the table is limited, we are forced to use M4R with $t = 1$, i.e., we use single 4-bit field elements as indices for the table and the result of the lookup is a single byte that corresponds to the result of two multiplications. Doing 32 of these lookups in parallel means we can do 64 field multiplications per `vpshufb` instruction. Our strategy is similar to the shuffle-based implementation of [BCH+23], with the difference that we lookup two multiplications instead of just one, which doubles the number of multiplications per `vpshufb` instruction.

The `vpshufb` instruction expects the 32 indices in the low nibbles of an AVX2 register, so, to multiply a vector of nibble-packed elements, we perform a lookup with the odd elements by masking out the high nibbles, and a lookup with the even elements after masking out the low nibbles and shifting down by four bits. The lookups result in a register that holds the products involving the odd nibbles, and another holding the products with even nibbles. Rather than interleaving them immediately, it is more efficient to accumulate the odd products and the even products separately and interleave the accumulated results *only once* at the end.

For setting up the multiplication tables, we use the fast method described in [BCH+23]. Since we do two multiplications per lookup, we use their method twice, and interleave the tables. In Verify we use a faster, variable-time method that avoids on-the-fly computation and uses index-dependent table lookups of precomputed tables instead.

The Intel Skylake architecture processes one `vpshufb` per cycle and the Ice Lake architecture two `vpshufb` instructions per cycle, both with one cycle latency. The products are accumulated using `vpxor` which has a throughput of three instructions per cycle. On Skylake, assuming everything can be pipelined perfectly, we expect to be bottlenecked only by the `vpshufb` instructions and an upper limit of 64 multiply-and-accumulate operations per cycle. On Ice Lake, three ports can handle `vpxor` and `vpshufb` instructions. Every 64 multiplications generate two micro-operations (one `vpxor` and one `vphufb`), and we can handle three of these micro-operations per cycle, so we expect an upper limit of $64 \cdot 3/2 = 96$ multiply-and-accumulate operations per cycle. Our experimental results are close to these theoretical upper limits.

Vectorization for the parameter sets. MAYO allows very natural vectorization since most arithmetic in $\mathbb{F}_{16}$ occurs m times independently. In MAYO_1 and MAYO_2 with $m = 64$,

the parallel operations fit in one AVX2 vector. In MAYO_3 with $m = 96$ and MAYO_5 with $m = 128$, they occupy two AVX2 vectors. In the case of MAYO_3, we overlap two vectors, which duplicates 32 operations and allows to easily extend the method to values of m that are not a multiple of 64.

MAYO matrix multiplications with AVX2. We consider groups of matrix multiplications as they occur in KeyGen, Sign, and Verify.

- KeyGen: Computing $-\mathbf{O}^\mathsf{T}(\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{P}_i^{(2)})$ consists of two matrix multiplications: $\mathbf{P}_i^{(1)}\mathbf{O}$ with upper triangular $\mathbf{P}_i^{(1)}$ followed by $\mathbf{O}^\mathsf{T}$ multiplied by the resulting product. Only the multiplication tables of $\mathbf{O}$ are needed for the multiplications. The code for $\mathbf{P}_i^{(1)}\mathbf{O}$ is shown in Figure 3. The computation consists of applying shuffle and xor operations $\frac{v^2o}{2}$ times in interleaved form, and de-interleaving the results at the end of each linear combination. For this, we use in total $\frac{v^2o}{2}$ `vpshufb` and `vpxor` instructions for multiply-and-accumulate, for de-interleaving $5vo$ `vpxor`, vo `vpand`/`vpsrlw`/`vpsllw` instructions, and v^2 `vpsrlw` and $2v^2$ `vpand` instructions for extracting the nibbles of $\mathbf{P}_i^{(1)}$.
- Sign: The three matrix multiplications for $\mathbf{V} \cdot \mathbf{P}_i^{(1)} \cdot \mathbf{V}^\mathsf{T}$ and $\mathbf{V} \cdot \mathbf{L}_i$ can be grouped using only the multiplication tables for the upper triangular $\mathbf{V} \in \mathbb{F}_q^{v \times k}$.
- Verify: The five matrix multiplications involved in $\mathbf{S}\begin{pmatrix}\mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)}\end{pmatrix}\mathbf{S}^\mathsf{T}$ are computed using only the multiplication tables of triangular $\mathbf{S} \in \mathbb{F}_q^{n \times k}$. The two matrix multiplications in $\mathbf{S}^{(1)}\mathbf{P}_i^{(1)} + \mathbf{S}^{(2)}\mathbf{P}_i^{(2)}$ are combined in a single function which allows to do the de-interleaving only once.

5.2 Arm Cortex-M4

M4R for MAYO on Cortex-M4. We consider the use for of M4R the three largest matrix multiplications in MAYO: $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)T})\mathbf{O}$ and $\mathbf{P}_i^{(1)}\mathbf{V}^\mathsf{T}$ in Sign, and $\mathbf{P}_i^{(1)}\mathbf{O}$ in KeyGen. Since $\mathbf{P}_i^{(1)}$ are public matrices, we can use M4R without any timing side-channel concerns. We compute $(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)T})\mathbf{O}$ as $\mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{P}_i^{(1)T}\mathbf{O}$ in order to not have to expand to a full square matrix. However, we do both operations at same time to avoid recomputing the linear combinations of the stripes of $\mathbf{O}$. Since two $\mathbb{F}_{16}$ elements are packed into one byte, it appears natural to use $t = 2$. This results in look-up tables of $256 \cdot k$ bytes (15 to 32 KB), which we consider acceptable. Using $t = 3$ seems infeasible as it would require multiple hundred KB of look-up tables.

We have to overcome three obstacles to apply M4R (as presented in subsection 2.2) to these multiplications:

1. $\mathbf{P}_i^{(1)}$ is an upper-triangular matrix, which means we require M4R algorithm for both upper ($\mathbf{P}_i^{(1)}$) and lower ($\mathbf{P}_i^{(1)T}$) triangular matrices. In the tail (head) of each stripe of the upper (lower) triangular matrix, one has to pad with a zero accordingly. We do this on-the-fly.
2. $\mathbf{P}_i^{(1)}$ is stored as a column-major Macaulay matrix, which means that the elements of the rows of each stripe of the matrix are not stored consecutively. We considered changing the order of the elements (in the specification). There are, however, three reasons against doing so: (1) There appears to be no representation that works well for reading from $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(1)\mathsf{T}}$ at the same time; (2) A different representation

Algorithm 7 Matrix multiplication using M4R and illustrated for the $\mathbf{P}_i^{(1)}\mathbf{O}$ batched matrix multiplication. $\mathbf{A}[\mathrm{r}, \mathrm{c}]$ refers to the element in row r in column c. Note the different representations of inputs and outputs. $\mathbf{P}_i^{(1)}$ is in column-major Macaulay form, i.e., using (row, column, batch) indexing with two elements stored in one byte. $\mathbf{PO}_i$ is using (row, batch, column) indexing with 8 elements stored in one `uint32_t`. If o is not divisible by 8, we pad with zeros accordingly.

Input $\mathbf{P}_i^{(1)}$: m upper triangular matrices of dimension $v \times v$
Input $\mathbf{O}$: matrix of dimension $v \times o$
Input/Output $\mathbf{PO}_i$: m upper triangular matrices of dimension $v \times o$

$\mathrm{o_{u32}} \leftarrow \lceil \frac{\mathrm{o}}{8} \rceil$
`uint32_t` table$[\mathrm{o_{u32}} \cdot 256]$
`uint32_t` rows$[\mathrm{o_{u32}} \cdot 8]$
for col $\leftarrow 0$ **to** v **by** 2 **do**
  table $\leftarrow 0$
  rows $\leftarrow 0$
  for $i \leftarrow 0$ **to** o **do** // `Pack first and second row of stripe`
    rows$[i/8] =$ rows$[i/8] \oplus (\mathbf{O}[\mathrm{col}, i]) \ll (4 \cdot (i\%8))$
    rows$[\mathrm{o_{u32}} \cdot 4 + (i/8)] =$ rows$[\mathrm{o_{u32}} \cdot 4 + (i/8)] \oplus (\mathbf{O}[\mathrm{col}+1, i]) \ll (4 \cdot (i\%8))$
  for $i \leftarrow 0$ **to** $\mathrm{o_{u32}}$ **do** // `Multiply each element of rows by` x, x^2, x^3
    rows$[\mathrm{o_{u32}} + i] =$ rows$[i] \cdot x$
    rows$[2 \cdot \mathrm{o_{u32}} + i] =$ rows$[\mathrm{o_{u32}} + i] \cdot x$
    rows$[3 \cdot \mathrm{o_{u32}} + i] =$ rows$[2 \cdot \mathrm{o_{u32}} + i] \cdot x$
    rows$[5 \cdot \mathrm{o_{u32}} + i] =$ rows$[4 \cdot \mathrm{o_{u32}} + i] \cdot x$
    rows$[6 \cdot \mathrm{o_{u32}} + i] =$ rows$[5 \cdot \mathrm{o_{u32}} + i] \cdot x$
    rows$[7 \cdot \mathrm{o_{u32}} + i] =$ rows$[6 \cdot \mathrm{o_{u32}} + i] \cdot x$
  for $t \leftarrow 0$ **to** 7 **do** // `Compute all linear combinations of rows`
    for $i \leftarrow 0$ **to** $(1 \ll t)$ **do**
      for $j \leftarrow 0$ **to** $\mathrm{o_{u32}}$ **do**
        table$[(i + (1 \ll t)) \cdot \mathrm{o_{u32}} + j] =$ table$[i \cdot \mathrm{o_{u32}} + j] \oplus$ rows$[t \cdot \mathrm{o_{u32}} + j]$
  for row $\leftarrow 0$ **to** col **do** // `Process pairs of element`
    for $k \leftarrow 0$ **to** m **do**
      byte $= \mathbf{P}_k^{(1)}[\mathrm{row}, \mathrm{col}] + \mathbf{P}_k^{(1)}[\mathrm{row}, \mathrm{col}+1] \ll 4$
      for $j \leftarrow 0$ **to** $\mathrm{o_{u32}}$ **do**
        $\mathbf{PO}_k[\mathrm{row}, j] = \mathbf{PO}_k[\mathrm{row}, j] \oplus$ table$[\mathrm{o_{u32}} \cdot \mathrm{byte} + j]$
  for $k \leftarrow 0$ **to** m **by** 2 **do** // `Tail of stripe: pad with zero`
    byte $= \mathbf{P}_i^{(1)}[\mathrm{col}+1, \mathrm{col}+1] \ll 4$
    for $j \leftarrow 0$ **to** $\mathrm{o_{u32}}$ **do**
      $\mathbf{PO}_k[\mathrm{col}+1, j] = \mathbf{PO}_k[\mathrm{col}+1, j] \oplus$ table$[\mathrm{o_{u32}} \cdot \mathrm{byte} + j]$

```
static inline
void mayo_12_P1_times_O_avx2(const __m256i *P1, __m256i *O_multabs, __m256i *acc){
  const __m256i low_nibble_mask  = _mm256_set1_epi8(0x0f);
  for (size_t r = 0; r < V_PARAM; r++) {
    // do multiplications for one row and accumulate results in temporary format
    __m256i temp[O_PARAM] = {0};
    for (size_t c = r; c < V_PARAM; c++) {
      __m256i in_odd = _mm256_loadu_si256(P1++);
      __m256i in_even = _mm256_srli_epi16(in_odd, 4) & low_nibble_mask;
      in_odd &= low_nibble_mask;
      for (size_t k = 0; k < O_PARAM; k+=2) {
        temp[k]     ^= _mm256_shuffle_epi8(O_multabs[O_PARAM/2*c + k/2], in_odd);
        temp[k + 1] ^= _mm256_shuffle_epi8(O_multabs[O_PARAM/2*c + k/2], in_even);
      }
    }
    // convert to normal format and add to accumulator
    for (size_t k = 0; k < O_PARAM; k+=2) {
      __m256i t = (temp[k + 1] ^ _mm256_srli_epi16(temp[k],4)) & low_nibble_mask;
      acc[(r*O_PARAM) + k] ^= temp[k] ^ _mm256_slli_epi16(t,4);
      acc[(r*O_PARAM) + k + 1] ^= temp[k+1] ^ t;
      }
  }
}
```

Figure 3: C code with compiler intrinsics for computing $\mathbf{P}_i^{(1)}\mathbf{O}$ in KeyGen for MAYO_1 and MAYO_2.

would drastically slow down implementations using different multiplication methods (such as our AVX2 implementations); (3) Changing it to a stripe-wise representation would likely force many platforms to use M4R with the parameterization chosen in this paper, which we deem undesirable. We, hence, decided to stick with the more standard column-major Macaulay matrix and perform the address computations and assembly of the row of the stripe on-the-fly.

3. The table look-ups result in rows that have to be accumulated to the resulting matrix: those elements are not stored consecutively in the canonical representation. Converting the representation on-the-fly results in very poor performance. We instead store the results as they are stored in the look-up table and merge the transformation of the representation into the addition following each of the multiplications. This results in competitive performance.

The process for $\mathbf{P}_i^{(1)}\mathbf{O}$ for $t = 2$ is outlined in Algorithm 7 and works analogously for other matrix multiplications.

Further matrix multiplications. There are three matrix multiplications ($\mathbf{O}^\mathsf{T} \cdot \circ$ in KeyGen, $\mathbf{V} \cdot \mathbf{L}_i$ and $\mathbf{V} \cdot \mathbf{P}_i^{(1)}\mathbf{V}$) for which we cannot use M4R due to timing side-channel concerns. In these cases, we make use of the bitsliced arithmetic and bitslice the inputs on-the-fly. This does come with some performance penalty (1.7 vs. 0.8 arithmetic instructions/field multiplication). However, the affected matrix multiplication generally involve matrices of relatively small dimension and, hence, this slow-down is outweighed by the performance gains of using M4R.

Verification. One could consider implementing verification using M4R as presented above. However, the trick presented in [CKY21] vastly outperforms the former idea. Hence, our verification stays almost the same as for the bitsliced variant. The only part that requires changes are the final multiplications and we can choose between the methods presented before (Figure 2). For the nibble-sliced representation, a multiplication by x requires 10 instructions (operating on 8 packed field elements), and, hence, Method 2 from Figure 2 performs better than Method 3. Note that from counting arithmetic instructions, it seems that the bitsliced variant performs much better than the nibble-sliced variant which suggests our proposed representation change would result in a significant slow-down compared to the bitsliced representation. This is, however, not the case: both variants (bitsliced representation using Method 3, nibble-sliced representation using Method 2) use around the same number of cycles for verification. This happens due to register pressure: when working on bitsliced field elements, one always has to work with 32 elements packed in 4 registers, while in the nibble-sliced variant, we can simply work on 8 elements in parallel. This allows for Method 2 to not require any spills to memory at all, which results in code competitive with the previous implementation.

6 Results

6.1 AVX2 Performance

We benchmarked the AVX2-optimized bitsliced and nibble-sliced (M4R) implementation on two Intel architectures: Skylake (Intel Xeon X3-1245 v5) and the more modern Ice Lake (Intel Xeon Gold 6338). The C code, using AVX2 compiler intrinsics, was compiled using clang-14 on Ubuntu 22.04.3 LTS. Turbo Boost was deactivated to achieve consistent timings. Our AES-CTR implementation is derived from libOQS [SM16] and achieves 0.63 cpb (Skylake), which comes close to the theoretical encryption-only limit of 0.625 cpb. On Ice Lake, the same implementation benefits from the double AES-NI throughput and achieves 0.32 cpb. Since SHAKE256 performance has only a marginal impact in MAYO, we use a plain non-optimized C implementation derived from PQClean [KSSW22].

Matrix multiplication. The results of the matrix multiplications that dominate the MAYO runtime are summarized in Table 3. The multiplication performance for the nibble-sliced implementation ranges between 45.6 - 56.5 mul/cycle (Skylake) and 65.0 - 78.8 mul/cycle (Ice Lake). The improvement on Ice Lake is due to the increased `vpshufd` throughput of 0.5 cpi compared to 1 cpi on Skylake. The multiplication throughput of MAYO_3 is about one fourth less than these numbers. Setting up the multiplication tables takes 5.2 cycles and 7.8 cycles per nibble on Ice Lake and Skylake, respectively. Setting up the multiplication tables in variable-time as used in verification takes only 1.1 cycles and 1.3 cycles per nibble on Ice Lake and Skylake, respectively. Our implementation reuses the multiplication tables for several matrix multiplications. Compared to the bitsliced implementation, the nibble-sliced matrix multiplications (including calculating multiplication tables) achieve a speedup of a factor between 3.6× and 5.9×.

Overall performance. The overall results are shown in Table 1. The nibble-sliced implementation using M4R leads to speedups between 2.0× and 3.6× compared to the bitsliced implementation. As AES-NI and `vpshufd` instructions are instrumental for the nibble-sliced performance, their increased throughput on Ice Lake leads to further speedups compared to the older Skylake architecture of up to 75% (KeyGen), 40% (Sign) and 79% (Verify). The fastest variant MAYO_1 on a single Ice Lake core at 2.0 GHz computes 45 924 KeyGen/sec, 9 162 signatures/sec and 37 272 verifications/sec. When reusing the

Table 1: Performance of MAYO in CPU cycles on Intel Xeon E3-1245 v5 (Skylake) and Xeon Gold 6338 (Ice Lake) using the bitsliced representation (round 1 specification) and the modified nibble representation.

Bitsliced Representation

	Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
Skylake	$MAYO_1$	159 186	212 208	44 058	589 202	213 716
	$MAYO_2$	424 894	437 778	59 288	690 878	135 820
	$MAYO_3$	835 694	1 380 698	147 912	2 816 584	908 390
	$MAYO_5$	1 806 558	3 204 710	355 200	5 755 844	1 483 332
Ice Lake	$MAYO_1$	110 338	162 064	22 380	459 614	148 250
	$MAYO_2$	310 166	342 212	30 256	540 018	94 876
	$MAYO_3$	511 526	629 052	74 988	1 676 162	612 806
	$MAYO_5$	1 209 482	1 995 956	180 692	3 978 970	1 158 326

Nibble Representation (M4R)

	Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
Skylake	$MAYO_1$	73 668	82 820	43 970	283 126	83 846
	$MAYO_2$	144 508	154 002	59 178	324 402	84 974
	$MAYO_3$	295 606	358 416	147 758	920 944	344 994
	$MAYO_5$	642 690	889 100	355 238	1 737 426	706 316
Ice Lake	$MAYO_1$	43 550	53 710	22 432	218 300	53 660
	$MAYO_2$	86 014	98 402	30 244	239 852	47 360
	$MAYO_3$	169 258	237 450	74 992	718 586	205 938
	$MAYO_5$	369 898	517 660	180 568	1 244 038	401 310

expanded keys, signing and verification can even perform 12 151 signatures/sec and 64 045 verifications/sec. All reported results are the median of 10 000 iterations.

Comparison with other schemes. A comparison of our MAYO implementation with other schemes, benchmarked on the same system, is shown in Table 2. The first candidate for comparison is OV [BCH+23]. When using compact keys, MAYO greatly outperforms OV by factors of 27× to 95× for KeyGen, factors of 6.5× to 15.1× for Sign, and factors of 3.1× to 4.3× for Verify. In cases that allow to store or re-use expanded keys, OV signing is 1.6× to 2.1× faster than MAYO. However, MAYO's Verify is 1.5× to 3.4× faster than OV at the same security level, and MAYO's expanded keys are much more compact than those of OV (e.g., 70 KB for MAYO and 278 KB for OV at SL I). Our MAYO implementation is competitive with the fastest lattice-based signature schemes. It outperforms Dilithium's KeyGen and Verify at security level 1, Sign is on par when using compact keys, and outperforms Dilithium when using pre-expanded keys. At security levels 3 and 5, Dilithium has a performance advantage especially in KeyGen and Sign. Overall, our MAYO implementation has balanced performance characteristics without big trade-offs between KeyGen, Sign, and Verify. Compared to OV, it has only moderate performance trade-offs when using compact keys.

6.2 Cortex-M4 Performance

This section presents the performance of our two implementations on an Arm Cortex-M4 microcontroller and compares the results to implementations of other post-quantum signature schemes. We target the ST NUCLEO-L4R5ZI development board with 640 KiB of RAM and 2 MiB of flash memory. We use the pqm4 [KPR+] library for benchmarking.

Table 2: MAYO performance in CPU cycles using AVX2 optimizations in comparison with other post-quantum signature schemes running on Intel Ice Lake (Xeon Gold 6330). Dilithium, Falcon and SPHINCS+ benchmarks use libOQS v0.9.0-rc1 with AVX2 optimized code.

Type	Sec. Lvl.	Key Gen.	Sign	Verify
MAYO [BCC+23] (default/pre-expanded)				
MAYO_1	1	$44k/44k$	$218k/165k$	$54k/31k$
MAYO_2	1	$86k/86k$	$240k/142k$	$47k/17k$
MAYO_3	3	$169k/169k$	$719k/481k$	$206k/131k$
MAYO_5	5	$370k/370k$	$1\,244k/726k$	$401k/221k$
Oil and Vinegar [BCH+23] (pkc+skc/classic)				
ovIp	1	$2\,316k/2\,341k$	$1\,548k/79k$	$168k/58k$
ovIs	1	$3\,715k/3\,734k$	$2\,063k/83k$	$203k/46k$
ovIII	3	$13\,168k/12\,832k$	$8\,293k/243k$	$679k/197k$
ovV	5	$34\,989k/35\,792k$	$18\,802k/462k$	$1\,514k/364k$
TUOV [DGG+] (pkc+skc/classic)				
tuov-Ip	1	$3\,262k/5\,916k$	$3\,639k/134k$	$241k/56k$
tuov-Is	1	$11\,797k/29\,323k$	$19\,607k/130k$	$273k/43k$
tuov-III	3	$16\,237k/29\,815k$	$18\,020k/354k$	$1\,107k/191k$
tuov-V	5	$38\,122k/68\,089k$	$40\,046k/637k$	$2\,947k/359k$
Dilithium [LDK+20]				
dilithium2	2	$81k$	$219k$	$79k$
dilithium3	3	$137k$	$355k$	$129k$
dilithium5	5	$212k$	$420k$	$204k$
Falcon [PFH+20]				
falcon-512	1	$20\,672k$	$705k$	$135k$
falcon-1024	5	$59\,019k$	$1\,427k$	$262k$
SPHINCS+ [HBD+20]				
sha256-128f-simple	1	$618k$	$14\,716k$	$1\,269k$
sha256-128s-simple	1	$39\,554k$	$298\,746k$	$517k$
sha256-192f-simple	3	$924k$	$25\,329k$	$2\,129k$
sha256-192s-simple	3	$58\,492k$	$563\,717k$	$983k$
sha256-256f-simple	5	$2\,412k$	$50\,912k$	$2\,240k$
sha256-256s-simple	5	$38\,076k$	$507\,125k$	$1\,295k$

For AES, we use the t-table implementation by Stoffelen and Schwabe [SS16] (as it is only used for expanding the public matrix). For SHAKE, we use the Armv7-M implementation in the XKCP [DHP+] by the Keccak team. Both implementations are also included in pqm4. We compile our code using the Arm GNU toolchain[4] Version `12.3Rel1`.

Our implementation requires to store the expanded secret key on the stack. For MAYO_5, this alone occupies 563 KB of memory leaving not enough space for other variables needed. Therefore, we focus on MAYO_1, MAYO_2, and MAYO_3 here as those fit the 640 KiB easily. Studying memory-optimized implementations of MAYO is promising future work, e.g., one could generate the coefficients of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ on the fly to avoid the memory cost of storing them.

Matrix multiplications. We first present results for the three matrix multiplications that are dominating the run-time of MAYO. Table 4 compares the performance of the 3

[4] https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads

Table 3: AVX2 performance in CPU cycles of core arithmetic involving the public key that can benefit from the method of the four Russians (M4R). Multiplication tables are reused among the group of matrix multiplications.

		$-\mathbf{O}^\mathsf{T}(\mathbf{P}_i^{(1)}\mathbf{O}-\mathbf{P}_i^{(2)})$ KeyGen		$\mathbf{V}\cdot\mathbf{P}_i^{(1)}\cdot\mathbf{V}^\mathsf{T}$ $\mathbf{V}\cdot\mathbf{L}_i$ Sign		$\mathbf{S}\begin{pmatrix}\mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)}\\ \mathbf{0} & \mathbf{P}_i^{(3)}\end{pmatrix}\mathbf{S}^\mathsf{T}$ Verify	
Skylake							
MAYO$_1$	bitsliced	107 107		162 638		158 138	
	M4R	22 494	(4.76 ×)	32 800	(4.96 ×)	31 478	(5.02 ×)
MAYO$_2$	bitsliced	352 497		84 338		64 556	
	M4R	75 072	(4.70 ×)	22 524	(3.74 ×)	17 928	(3.60 ×)
MAYO$_3$	bitsliced	684 302		957 721		735 788	
	M4R	133 855	(5.11 ×)	188 289	(5.09 ×)	181 499	(4.05 ×)
MAYO$_5$	bitsliced	1 412 844		1 657 854		1 095 647	
	M4R	278 519	(5.07 ×)	333 199	(4.98 ×)	331 529	(3.30 ×)
Ice Lake							
MAYO$_1$	bitsliced	83 336		122 832		122 849	
	M4R	17 237	(4.83 ×)	25 265	(4.86 ×)	23 117	(5.31 ×)
MAYO$_2$	bitsliced	268 767		65 373		60 830	
	M4R	47 943	(5.61 ×)	12 265	(5.33 ×)	12 100	(5.03 ×)
MAYO$_3$	bitsliced	426 969		615 090		511 093	
	M4R	86 050	(4.96 ×)	119 493	(5.15 ×)	118 403	(4.32 ×)
MAYO$_5$	bitsliced	1 022 887		1 200 700		904 729	
	M4R	177 396	(5.77 ×)	205 161	(5.85 ×)	203 966	(4.44 ×)

Table 4: Cortex-M4 Performance of core arithmetic involving the public key that can benefit from the method of the four Russians (M4R).

		$(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O}$ Sign		$\mathbf{P}_i^{(1)}\mathbf{V}^{\mathsf{T}}$ Sign		$\mathbf{P}_i^{(1)}\mathbf{O}$ KeyGen	
MAYO$_1$	bitsliced	2 165 337		1 323 797		1 177 752	
	M4R	1 244 009	(1.74 ×)	1 119 136	(1.18 ×)	714 332	(1.65 ×)
MAYO$_2$	bitsliced	5 199 607		629 400		2 830 681	
	M4R	2 906 460	(1.79 ×)	681 081	(0.92 ×)	1 683 616	(1.68 ×)
MAYO$_3$	bitsliced	9 535 835		5 635 495		5 126 000	
	M4R	6 576 258	(1.45 ×)	3 452 417	(1.63 ×)	3 525 668	(1.45 ×)

Table 5: Performance of MAYO on the Arm Cortex-M4 using the bitsliced representation (round 1 specification) and the modified nibble representation. Cycles presented are the average of 1000 executions.

Bitsliced Representation

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
MAYO$_1$	5 245 602	5 293 828	3 098 810	9 181 163	4 887 097
MAYO$_2$	11 925 123	9 418 744	4 149 233	12 042 353	5 103 785
MAYO$_3$	18 306 278	20 052 487	10 458 654	32 008 516	15 587 746

Nibble Representation (M4R)

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
MAYO$_1$	4 410 207	4 381 417	3 098 817	8 269 909	4 807 561
MAYO$_2$	8 846 960	7 154 898	4 149 239	9 915 805	5 101 410
MAYO$_3$	15 971 829	17 196 207	10 471 338	27 400 909	15 573 359

operations for each of the MAYO parameter sets. We see that the bitsliced implementation is significantly outperformed by M4R implementations except for one case, but in that case the gains for the first matrix multiplication outweigh the performance loss for the second.

MAYO performance. Table 5 contains the results for all algorithms MAYO signature scheme on the Cortex-M4. The change of representation and use of M4R result in speed-ups for KeyGen, ExpandSK, and Sign. For verification, the performance is almost the same for both representations as described in section 5.2.

Comparison to other PQC signatures. Table 6 compares the performance of our MAYO implementation on the Arm Cortex-M4 with the MAYO implementation[5] from [GMSS23], an FPGA implementation from [HSMR23], and implementations of other PQC schemes. Compared to the existing MAYO implementation from [GMSS23] (with very similar, but not identical parameters), our implementation outperforms signing by 12.9× and verification by 4.3×. There is an important difference between the two implementations: [GMSS23] does not correctly implement the linear equation solving. They instead use the approach

[5]We report the numbers obtained on the Arm Cortex-M4 as reported in https://github.com/mayo-pqm4/mayo-pqm4. These cycle counts are higher than those reported in the paper for the Cortex-M7.

Table 6: MAYO performance on Cortex-M4 in comparison to other post-quantum signature schemes optimized for different platforms. MAYO *pre* variants refer to pre-expanded public and secret keys in a similar fashion as *classic* OV. The implementation from [GMSS23] uses slightly different parameters ($n = 66, m = 64, o = 7, k = 10$) than MAYO_1– we call it MAYO_1 (*) in the table. The results presented from [HSMR23] are based on an FPGA implementation on a Xilinx Kintex-7 KC705 board clocked at 100 MHz.

Type	Sec. Level	Plat.	Key Gen.	Sign	Open
MAYO [BCC+23]					
MAYO_1	1	M4	4 410k	8 270k	4 808k
MAYO_1-pre	1	M4	4 410k	3 888k	1 709k
MAYO_2	1	M4	8 847k	9 916k	5 102k
MAYO_2-pre	1	M4	8 847k	2 761k	952k
MAYO_3	3	M4	15 972k	27 401k	15 573k
MAYO_3-pre	3	M4	15 972k	10 204k	5 102k
MAYO_1 (*) [GMSS23]	1	M4	—	50 183k	7 371k
MAYO_1 [HSMR23]	1	KC705	12 182	49 926	12 722
MAYO_3 [HSMR23]	3	KC705	38 325	137 358	39 740
Oil and Vinegar [BCH+23]					
`ovIp` (classic)	1	M4	138 833k	2 482k	995k
`ovIp` (pkc+skc)	1	M4	175 021k	88 757k	11 551k
`ovIs` (classic)	1	M4	195 744k	2 374k	616k
`ovIs` (pkc+skc)	1	M4	296 161k	113 446k	16 045k
Dilithium [AHKS22]					
dilithium2	2	M4	1 598k	4 093k	1 572k
dilithium3	3	M4	2 827k	6 623k	2 692k
Falcon [Por19]					
falcon-512	1	M4	163 994k	39 014k	473k
SPHINCS+ [KPR+]					
sha256-128f-simple	1	M4	15 388k	382 534k	21 151k
sha256-128s-simple	1	M4	985 367k	7 495 604k	7 166k
sha256-192f-simple	3	M4	22 646k	639 322k	32 940k
sha256-192s-simple	3	M4	1 450 073k	13 764 197k	11 764k

described in [CKY21] trying to achieve an upper triangular matrix with ones on the diagonal. However, in MAYO there are more variables than equations, and hence, we have to select one solution at random as described in section 3. The approach of [GMSS23] has two problems: (1) It does not select a solution uniformly at random. Instead, it selects solutions that have a higher-than-average number of zeros. This breaks the security proof of MAYO and can potentially lead to an attack; (2) While their approach is easier to implement and results in slightly better performance for a single iteration, it has a much higher failure probability of 1/15.

MAYO (as Oil-and-Vinegar) can benefit from pre-expanded public and secret keys. We report such variants in Table 6 (denoted by *pre*) to allow a fair comparison with the *classic* variant of Oil-and-Vinegar. We see that MAYO outperforms OV when using compressed public and secret keys, and comes very close to its performance when using pre-expanded keys. Due to the large cost of key expansion due to the high cost of AES, the performance of MAYO on the Cortex-M4 is not competitive with lattice-based signatures. When using pre-expanded keys, this difference vanishes. AES hardware acceleration or round-reduced AES (as proposed in [BCH+23]) would have a similar effect.

Acknowledgments

Matthias J. Kannwischer was supported by the Taiwan Ministry of Science and Technology through grant 109-2221-E-001-009-MY3, Academia Sinica Investigator Award AS-IA-109-M01, and the Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

References

[ADKF70] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and Igor Aleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[AH74] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education, 1974.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 853–871. Springer, Heidelberg, June 2022.

[arm] *Cortex-M4 Technical Reference Manual r0p0*. Available at https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC. Accessed Jan, 2024.

[BCC+23] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias Kannwischer. Mayo. MAYO specification, 2023. https://pqmayo.org/assets/specs/mayo.pdf.

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

[Beu22] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume

13203 of *LNCS*, pages 355–376. Springer, Heidelberg, September / October 2022.

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 33–48. Springer, Heidelberg, September 2009.

[CKY21] Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on cortex-M4. *IACR TCHES*, 2021(4):650–675, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9078.

[DCK+21] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. NIST PQC Standardization Process, 2021. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[DCP+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias J. Kannwischer, and Jacques Patarin. Rainbow. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[DGG+] Jintai Ding, Boru Gong, Hao Guo, Xiaoou He, Yi Jin, Yuansheng Pan, Dieter Schmidt, Chengdong Tao, Danli Xie, Bo-Yin Yang, and Ziyu Zhao. TUOV. Technical report, National Institute of Standards and Technology, 2022. https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[DHP+] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP.

[FG18] Ahmed Ferozpuri and Kris Gaj. High-speed FPGA implementation of the NIST round 1 rainbow signature scheme. In David Andrews, René Cumplido, Claudia Feregrino, and Dirk Stroobandt, editors, *2018 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2018, Cancun, Mexico, December 3-5, 2018*, pages 1–8. IEEE, 2018.

[GMSS23] Arianna Gringiani, Alessio Meneghetti, Edoardo Signorini, and Ruggero Susella. Mayo: Optimized implementation with revised parameters for armv7-m. Cryptology ePrint Archive, Paper 2023/540, 2023. https://eprint.iacr.org/2023/540.

[HBD+] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. Sphincs+. Technical report, National Institute of Standards and Technology, 2019. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[HBD+20] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl,

Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[HSMR23] Florian Hirner, Michael Streibl, Ahmet Can Mert, and Sujoy Sinha Roy. A hardware implementation of MAYO signature scheme. *IACR Cryptol. ePrint Arch.*, page 1267, 2023.

[int] *Intel Instruction Set Architecture*. Available at https://www.intel.com/content/www/us/en/developer/tools/isa-extensions/overview.html. Accessed Jan, 2024.

[KKS+21] Hyeokdong Kwon, Hyunjun Kim, Minjoo Sim, Wai-Kong Lee, and Hwajeong Seo. Look-up the rainbow: Efficient table-based parallel implementation of rainbow signature on 64-bit armv8 processors. *IACR Cryptol. ePrint Arch.*, page 1015, 2021.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 206–222. Springer, Heidelberg, May 1999.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[Lan10] Adam Langley. *ctgrind*, 2010. Available at https://github.com/agl/ctgrind. Accessed Jan, 2024.

[LDK+] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancréde Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-dilithium. Technical report, National Institute of Standards and Technology, 2019. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[LDK+20] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[NIS22] NIST Computer Security Division. Post-Quantum Cryptography: Digital Signature Schemes, 2022. https://csrc.nist.gov/projects/pqc-dig-sig.

[oST] National Institute of Standards and Technology. Post-quantum cryptography. NIST PQC Standardization Process. https://csrc.nist.gov/projects/post-quantum-cryptography.

[oST22] National Institute of Standards and Technology. Selected algorithms 2022. NIST PQC Standardization Process, 2022. https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022.

[Pat95] Jacques Patarin. Cryptanalysis of the Matsumoto and Imai public key scheme of eurocrypt'88. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 248–261. Springer, Heidelberg, August 1995.

[Pat97] Jacques Patarin. The Oil and Vinegar signature scheme. Dagstuhl Workshop on Cryptography, 1997.

[Pet13] Albrecht Petzoldt. Hybrid approach for the fast verification for improved versions of the UOV and rainbow signature schemes. *IACR Cryptol. ePrint Arch.*, page 315, 2013.

[PFH+] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. Technical report, National Institute of Standards and Technology, 2019. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[PFH+20] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[Por19] Thomas Pornin. New efficient, constant-time implementations of Falcon. Cryptology ePrint Archive, Report 2019/893, 2019. https://eprint.iacr.org/2019/893.

[SM16] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 14–37. Springer, Heidelberg, August 2016.

[SMA+23] Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. Hamayo: A reconfigurable hardware implementation of the post-quantum signature scheme MAYO. *IACR Cryptol. ePrint Arch.*, page 1135, 2023.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 180–194. Springer, Heidelberg, August 2016.