Round 2 Version

Submitters

Ward Beullens
Fabio Campos
Sofía Celi
Basil Hess
Matthias J. Kannwischer

Contact

`contact@pqmayo.org`
`https://pqmayo.org/`

Chapter 1

Introduction and design rationale

This document presents a detailed description of *MAYO*, a multivariate quadratic signature scheme, which was introduced by Beullens in [Beu22]. It is a variant of the *Oil and Vinegar* signature scheme proposed in 1997 by Patarin [Pat97].

Oil and Vinegar. Since 1985, various authors have proposed building public key schemes where the public key is a set of multivariate quadratic equations over a small finite field K. The general problem of solving such a set of equations is NP-hard and considered a good basis for post-quantum cryptography. The *Oil and Vinegar* scheme (sometimes referred to as *unbalanced Oil and Vinegar*) [KPG99, Pat97] is one of the earliest signature schemes in this framework, and has withstood the test of time remarkably well, despite considerable cryptanalytic efforts. It has very small signature sizes and good performance but suffers from somewhat larger public keys.

In the *Oil and Vinegar* scheme, the public key represents a trapdoored homogeneous multivariate map $\mathcal{P}(\mathbf{x}) = (p_1, \ldots, p_m) : \mathbb{F}_q^n \to \mathbb{F}_q^m$ which consists of a sequence of m multivariate quadratic polynomials $p_1(\mathbf{x}), \cdots, p_m(\mathbf{x})$ in n variables $\mathbf{x} = (x_1, \cdots, x_n)$. The trapdoor information is a secret subspace $O \subset \mathbb{F}_q^n$ of dimension m, on which $\mathcal{P}(\mathbf{x})$ evaluates to zero. Given a salted hash digest $\mathbf{t} \in \mathbb{F}_q^m$ of a message M, the trapdoor information allows sampling a signature $\mathbf{s}$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$.

To do this, the signer first picks a random vector $\mathbf{v} \in \mathbb{F}_q^n$, and then solves for a vector $\mathbf{o}$ in the oil space O such that $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{t}$. In general, for a quadratic maps $\mathcal{P}$ we can define its differential $\mathcal{P}'$ as $\mathcal{P}'(\mathbf{x}, \mathbf{y}) := \mathcal{P}(\mathbf{x} + \mathbf{y}) - \mathcal{P}(\mathbf{x}) - \mathcal{P}(\mathbf{y})$, which is a bilinear map. Using $\mathcal{P}'$, it becomes apparent that solving for $\mathbf{o}$ is easy, because

$$\mathcal{P}(\mathbf{v} + \mathbf{o}) = \underbrace{\mathcal{P}'(\mathbf{v}, \mathbf{o})}_{\text{Linear in } \mathbf{o}} + \underbrace{\cancel{\mathcal{P}(\mathbf{o})}}_{=0} + \underbrace{\mathcal{P}(\mathbf{v})}_{\text{fixed}} = \mathbf{t}$$

is a system of m linear equations in m variables (since O has dimension m). The signer outputs the signature $\mathbf{s} = \mathbf{v} + \mathbf{o}$. To verify a signature, the verifier simply recomputes $\mathcal{P}(\mathbf{s})$ and the hash digest $\mathbf{t}$, and verifies that they are equal.

One practical drawback of the scheme is that the public map $\mathcal{P}$ consists of approximately $mn^2/2$ coefficients. We can sample $\mathcal{P}$ such that approximately $m(n^2 - m^2)/2$ of the coefficients can be expanded publicly from a short seed, but the remaining $m^3/2$ coefficient still make for a relatively large public key size. (e. g., 66 KB for 128 bits of security). This problem is solved by the scheme we present in this document: MAYO.

MAYO rationale. MAYO is a variant of the *Oil and Vinegar* scheme whose public keys are smaller. A MAYO public key $\mathcal{P}$ has the same structure as an *Oil and Vinegar* public key, except 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. The advantage of this is that the problem of recovering O from $\mathcal{P}$ becomes much harder, which allows for smaller parameters. The reader can imagine O as being a needle that sits in the haystack $\mathbb{F}_q^n$. If the needle becomes smaller, then the haystack is allowed to be smaller as well, and the search problem remains difficult. However, since O is "too small", the algorithm to sample a signature $\mathbf{s}$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$ breaks down: the system $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{t}$ is now a system of m linear equations in only o variables, so it is very unlikely to have any solutions. We need a new way to produce and verify signatures.

The solution is to publicly "whip up" the oil and vinegar map $\mathcal{P}(\mathbf{x}) : \mathbb{F}_q^n \to \mathbb{F}_q^m$ into a k-fold larger map $\mathcal{P}^*(\mathbf{x}_1, \dots, \mathbf{x}_k) : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$, where k is a parameter of the scheme. The whipped map $\mathcal{P}^*$ is constructed in such a way that it evaluates to zero on the subspace $O^k = \{(\mathbf{o}_1, \dots, \mathbf{o}_k) \,|\, \forall i : \mathbf{o}_i \in O\}$ which has dimension ko. Concretely, we define:

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

where the $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ are fixed public matrices[1] (referred to as $\mathbf{E}$-matrices), and $\mathcal{P}'(\mathbf{x}, \mathbf{y})$, the differential of $\mathcal{P}$, is defined as $\mathcal{P}'(\mathbf{x}, \mathbf{y}) := \mathcal{P}(\mathbf{x}+\mathbf{y}) - \mathcal{P}(\mathbf{x}) - \mathcal{P}(\mathbf{y})$. We choose parameters such that $ko > m$ to make sure that the space O^k is large enough so that the signer can sample signatures $\mathbf{s} = (\mathbf{s}_1, \cdots, \mathbf{s}_k)$ such that $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}$ with the usual *Oil and Vinegar* approach. The signer first samples $(\mathbf{v}_1, \dots, \mathbf{v}_k) \in \mathbb{F}_q^{kn}$ at random, and then solves for $(\mathbf{o}_1, \dots \mathbf{o}_k) \in O^k$ such that

$$\mathcal{P}^*(\mathbf{v}_1 + \mathbf{o}_1, \dots, \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t}$$

which is a system of m linear equations in ko variables.

This approach drastically reduces the public key size, since all but approximately $mo^2/2$ coefficients of $\mathcal{P}$ can be expanded publicly from a short seed. For example, one of the parameter sets we propose for NIST security level 1 is $(n, m, o, k, q) = (81, 64, 17, 4, 16)$, which results in a public key of just 4912 bytes, and a signature size of 186 bytes (162 bytes for $\mathbf{s}$ and 24 bytes for the salt). Compared to the compressed *Oil and Vinegar* scheme at the same security level, this is a 14-fold reduction in public key size at the cost of a 2-fold increase in signature size. We also propose a parameter set with even smaller public keys, but larger signatures.

[1] For security reasons, we choose these matrices to have the property that all their non-trivial linear combinations have rank m.

Change Log

Modifications in Second Round Submission.

- **Different representation of sequences of $\mathbf{m}$ matrices.** Batches of matrices are now stored in nibble-sliced form, rather than bitsliced form. This change allows for significantly faster implementations on AVX2 and Arm NEON platforms by using shuffle instructions, and slightly more efficient implementations on Cortex-M4 platforms using the method of the four Russians. For more information, we refer to [BCC+24].
- **Updated security analysis.** The security analysis section was expanded to keep it up to date with the cryptanalysis of OV-based schemes. A paragraph about the rectangular minrank attack was added, the section about claw-finding attacks was expanded, and the system-solving section was expanded to encompass Hashimoto's algorithm for solving underdetermined systems of multivariate quadratic polynomials.
- **Updated parameters.** New parameter sets are selected satisfying the following criteria:
  - $\mathbf{n} - \mathbf{o} \leq \mathbf{m}$. In the round 1 submission, the parameters satisfied $o > n - m$, which means that the dimension of the variety defined by $\mathcal{P}(\mathbf{x}) = 0$ is larger than $n - m$, the generic dimension of a variety defined by m multivariate quadatic equations in n variables. We are not aware of ways to compute this dimension that are more efficient than known key recovery attacks, so to the best of our knowledge this does not break the Oil and Vinegar assumption, which says that $\mathcal{P}$ is computationally indistinguishable from a randomly chosen sequence of multivariate quadratic equations. Nevertheless, we decided to pick parameters with $o \leq n - m$, so that $\mathcal{P}(\mathbf{x}) = 0$ has dimension $n - m$. Additionally, this blocks the rectangular Minrank attack [FI23], which simplifies the concrete security analysis of MAYO.
  - **Increased security margin against system-solving attacks.** To hedge against improvements in generic system-solving methods, we pick parameters to have at least 10, 15, and 20 bits of security margin, for the NIST security level 1, 3, and 5 parameters respectively.
  - **Higher restart probability.** During the signing procedure, there is a small probability that the SampleSolution subroutine fails, in which case signing restarts. With the round 1 parameters this restart probability was lower than 2^{-36}, which makes it hard to cover the complete implementation with known answer tests. Therefore, for the round 2 parameters, we increased this probability to between 2^{-12} and 2^{-20}, so that the restarting is easier to test, but still low enough so that the average signing time is not affected much.
- **Added more implementations and benchmarks.** We have added an Arm NEON optimized implementation of MAYO to the submission package, and included benchmark results for the Apple M1 and M3 processors. We extended the Cortex-M4 optimized implementation to cover MAYO_3, and added benchmark results.

Differences between first round submission and [Beu22].

- **New parameter choices.** We propose new parameter choices that allow for simple and optimized implementations of MAYO. In particular, we choose to work over the finite field of order 16.
- **Instantiating the $\mathbf{E}$-matrices.** MAYO requires a set of $k(k+1)/2$ public matrices $\mathbf{E}_{i,j} \in \mathbb{F}_q^{m \times m}$ with the property that certain non-trivial linear combinations of them have rank m. We instantiate these matrices by the matrix representation of multiplication by $z^0, z^1, \ldots, z^{k(k+1)/2-1}$ in the field $\mathbb{F}_q[z]/(f(z))$ for some irreducible polynomial $f(z)$ of degree m. This is possible because we choose parameters where $k(k+1)/2 \leq m$.

Chapter 2

The MAYO protocol specification

2.1 Written specification

This section specifies the MAYO protocol. The set of public parameters for MAYO is defined in 2.1.7. The necessary notation and preliminaries are defined in 2.1.2. The encoding functionality is defined in 2.1.4. The signature functionality is defined in 2.1.5.

2.1.1 Parameters

The MAYO digital signature algorithm is parameterized by the following values:

– q, the size of a finite field $\mathbb{F}_q$. In this specification, we fix $q = 16$.

– m, the number of multivariate quadratic polynomials in the public key. We choose it to be even.

– n, the number of variables in the multivariate quadratic polynomials in the public key.

– o, the dimension of the oil space O.

– k, the whipping parameter, satisfying $k < n - o$.

– salt_bytes, the number of bytes in salt.

– digest_bytes, the number of bytes in the hash digest of a message.

– pk_seed_bytes, the number of bytes in $\mathsf{seed}_{\mathsf{pk}}$.

– $f(z) \in \mathbb{F}_q[z]$, an irreducible polynomial of degree m that does not divide the determinant of:

$$\mathbf{Z}^{(k\times k)} = \begin{pmatrix} z^{k-1} & z^{k-2} & \dots & z & 1 \\ z^{k-2} & z^{2k-2} & \dots & z^{k+1} & z^{k} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ z & z^{k+1} & \dots & z^{k(k+1)/2-2} & z^{k(k+1)/2-3} \\ 1 & z^{k} & \dots & z^{k(k+1)/2-3} & z^{k(k+1)/2-1} \end{pmatrix} \in \mathbb{F}_q[z]^{k\times k}.$$

The matrix $\mathbf{Z}^{(k\times k)}$ is symmetric, and the upper diagonal part contains the first $k(k+1)/2$ powers of z, ordered from left to right, top to bottom.

These parameters define the following values:

– sk_seed_bytes $=$ R_bytes $=$ salt_bytes: The length of R and of $\mathsf{seed}_{\mathsf{sk}}$, which are the same as that of salt.

– O_bytes $= \lceil (n-o)o/2 \rceil$: The number of bytes to represent the $\mathbf{O}$ matrix.

- v_bytes $= \lceil (n-o)/2 \rceil$: The number of bytes to store vinegar variables.
- P1_bytes $= m\binom{n-o+1}{2}/2$: The number of bytes to represent the $\{\mathbf{P}_i^1\}_{i\in[m]}$ matrices.
- P2_bytes $= m(n-o)o/2$: The number of bytes to represent the $\{\mathbf{P}_i^2\}_{i\in[m]}$ matrices.
- P3_bytes $= m\binom{o+1}{2}/2$: The number of bytes to represent the $\{\mathbf{P}_i^3\}_{i\in[m]}$ matrices.
- L_bytes $= m(n-o)o/2$: The number of bytes to represent the $\{\mathbf{L}_i\}_{i\in[m]}$ matrices.
- csk_bytes $=$ sk_seed_bytes: The number of bytes in the compact representation of a secret key.
- esk_bytes $=$ sk_seed_bytes $+$ O_bytes $+$ P1_bytes $+$ L_bytes: The number of bytes in the expanded representation of a secret key.
- cpk_bytes $=$ pk_seed_bytes $+$ P3_bytes: The number of bytes in the compact representation of a public key.
- epk_bytes $=$ P1_bytes $+$ P2_bytes $+$ P3_bytes: The number of bytes in the expanded representation of a public key.
- sig_bytes $= \lceil nk/2 \rceil +$ salt_bytes: The number of bytes in a signature.
- $\mathbf{E} \in \mathbb{F}_q^{m\times m}$, the matrix that corresponds to multiplication by z mod $f(Z)$.

2.1.2 Preliminaries and notation

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 $\{0,\dots,k-1\}$. We denote by $\{x_i\}_{i\in[k]}$ a sequence of objects $x_0,\dots,x_{k-1}$ indexed by elements of $[k]$. We denote the base-2 logarithm by $\log$, and we denote binomial coefficients by $\binom{n}{k}$, i.e., $\binom{n}{k} = n!/k!(n-k)!$.

Bytes and byte strings. Inputs and outputs to all MAYO API functions are byte strings. We denote by $\mathcal{B} = [256] = \{0,\dots,255\}$ the set of all bytes, i.e. 8-bit unsigned integers. By $\mathcal{B}^k$ we denote the set of zero-indexed byte strings of length k, and by $\mathcal{B}^*$ the set of byte strings of arbitrary length. For $a \in \mathcal{B}^{n_a}$ and $b \in \mathcal{B}^{n_b}$, we denote by $a \parallel b$ the concatenation of the strings, which result is an element of $\mathcal{B}^{n_a+n_b}$. If a is a byte string, we denote by $a[x:y]$ the substring starting with the x-th byte and ending with the $(y-1)$-th byte (inclusive), e. g., $a[0:10]$ consists of the first 10 bytes of a.

The field $\mathbb{F}_{16}$ and vectors over $\mathbb{F}_{16}$. We denote by $\mathbb{F}_{16}$ a finite field with 16 elements, which we represent concretely as $\mathbb{Z}_2[x]/(x^4+x+1)$. We denote the addition and multiplication of field elements a and b as $a+b$ and ab respectively, and we denote the multiplicative inverse of a as a^{-1}. We denote by $\mathbb{F}_{16}^n$ the set of vectors of length n over $\mathbb{F}_{16}$, i.e. lists of field elements of length n. If $\mathbf{x} \in \mathbb{F}_{16}^n, \mathbf{y} \in \mathbb{F}_{16}^n$, and $a \in \mathbb{F}_{16}$, we denote by $\mathbf{x}[i]$ or x_i the i-th entry of $\mathbf{x}$, i.e. $\mathbf{x} = \{x[i]\}_{i\in[n]} = \{x_i\}_{i\in[n]}$. For $0 \le i < j \le n$, we denote by $\mathbf{x}[i:j] \in \mathbb{F}_{16}^{j-i}$ the vector whose $j-i$ elements are $\mathbf{x}_i,\dots,\mathbf{x}_{j-1}$. We define the component-wise sum as $\mathbf{x}+\mathbf{y} := \{x_i + y_i\}_{i\in[n]}$, and the scalar multiplication as $a\mathbf{x} := \{ax_i\}_{i\in[n]}$.

Matrices and Matrix arithmetic. We denote by $\mathbb{F}_{16}^{m\times n}$ the set of (zero-indexed) matrices over $\mathbb{F}_{16}$ 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 $\mathbf{A} \in \mathbb{F}_{16}^{n\times n}$ is upper triangular if $\mathbf{A}[i,j] = 0$ for all $0 \le j < i < n$.

If $\mathbf{A} \in \mathbb{F}_{16}^{m\times n}$ and $\mathbf{B} \in \mathbb{F}_{16}^{m\times n}$ are matrices of the same size, then we denote their (entry-wise) sum by $\mathbf{A}+\mathbf{B}$. If $\mathbf{A} \in \mathbb{F}_{16}^{m\times n}$ and $\mathbf{B} \in \mathbb{F}_{16}^{n\times k}$, then we denote the matrix product by $\mathbf{AB}$, i.e. $\mathbf{AB} \in \mathbb{F}_{16}^{n\times k}$ is the matrix whose entry in row i and column j is equal to $\sum_{l=0}^{n} \mathbf{A}[i,l]\mathbf{B}[l,j]$. We denote by $\mathbf{A}^\intercal$ the transpose of $\mathbf{A}$, i.e. the matrix in $\mathbb{F}_{16}^{n\times m}$ such that $\mathbf{A}^\intercal[i,j] = \mathbf{A}[j,i]$ for all $0 \le j < m$ and $0 \le i < n$.

We define the function Upper : $\mathbb{F}_q^{n\times n} \to \mathbb{F}_q^{n\times n}$ that takes a square matrix $\mathbf{M}$ as input, and outputs the upper triangular matrix $\mathsf{Upper}(\mathbf{M})$, defined as $\mathsf{Upper}(\mathbf{M})_{ii} = \mathbf{M}_{ii}$ and $\mathsf{Upper}(\mathbf{M})_{ij} = \mathbf{M}_{ij} + \mathbf{M}_{ji}$ for $i < j$.

Sequences of m (upper triangular) matrices. The public keys and secret keys of MAYO contain sets of m (sometimes upper triangular) matrices. Concretely, we will encounter:

- $\mathbf{P}^{(1)} = \{\mathbf{P}_i^{(1)}\}_{i\in[m]}$, a sequence of m upper triangular matrices $\mathbf{P}_i^{(1)} \in \mathbb{F}_{16}^{(n-o)\times(n-o)}$.
- $\mathbf{P}^{(2)} = \{\mathbf{P}_i^{(2)}\}_{i\in[m]}$, a sequence of m matrices $\mathbf{P}_i^{(2)} \in \mathbb{F}_{16}^{(n-o)\times o}$.
- $\mathbf{P}^{(3)} = \{\mathbf{P}_i^{(3)}\}_{i\in[m]}$, a sequence of m upper triangular matrices $\mathbf{P}_i^{(3)} \in \mathbb{F}_{16}^{o\times o}$.
- $\mathbf{L} = \{\mathbf{L}_i\}_{i\in[m]}$, a sequence of m matrices $\mathbf{L}_i \in \mathbb{F}_{16}^{(n-o)\times o}$.

Sampling a solution to a system of linear equations. For $\mathbf{A} \in \mathbb{F}_q^{m\times ko}$ a matrix of rank m with $ko \ge m$, for $\mathbf{y} \in \mathbb{F}_q^m$ and $\mathbf{r} \in \mathbb{F}_q^{ko}$, the function SampleSolution$(\mathbf{A},\mathbf{y},\mathbf{r})$ (see Algorithm 2) outputs a solution $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{y}$. The solution space has dimension $ko - m$, and the random vector $\mathbf{r} \in \mathbb{F}_q^{ko}$ is used to pick each of the q^{ko-m} solutions with equal probability. This is done by solving the related system $\mathbf{A}\mathbf{x}' = y - \mathbf{A}r$ with the usual Gaussian Elimination approach, and outputting $\mathbf{x} = \mathbf{x}' + \mathbf{r}$. If the input matrix $\mathbf{A}$ does not have rank m, then SampleSolution$(\mathbf{A},\mathbf{y},\mathbf{r})$ outputs $\perp$, even in the somewhat unlikely case that the system $\mathbf{Ax} = \mathbf{y}$ has solutions. SampleSolution uses a subroutine EF (see Algorithm 1) that performs elementary row operations on an input matrix $\mathbf{B} \in \mathbb{F}_q^{m\times(n+1)}$ to put it in echelon form with leading terms equal to one. That is, the output of EF$(\mathbf{B})$ is a matrix where all the zero rows are at the bottom, the first non-zero element of each row is 1, and for all $i > 0$ the first non-zero element of row i is strictly to the right of the first non-zero element of row $i - 1$.

Algorithm 1 EF$(\mathbf{B})$

Input: A matrix $\mathbf{B} \in \mathbb{F}_q^{m\times(ko+1)}$
Output: A matrix $\mathbf{B}' \in \mathbb{F}_q^{m\times(ko+1)}$, the echelon form with leading ones of $\mathbf{B}$.

```
1: pivot_row ← $0$, pivot_column ← $0$
2: while pivot_row $< m$ and pivot_column $< ko + 1$ do
3:     possible_pivots ← $\{i \mid$ pivot_row $\le i < m$ and $\mathbf{B}[i,$ pivot_column$] \ne 0\})$
4:     if possible_pivots $= \emptyset$ then
5:         pivot_column ← pivot_column $+ 1$          // Move to next column if there is no pivot.
6:         continue
7:     next_pivot_row ← $\min$(possible_pivots)
8:     Swap($\mathbf{B}$[pivot_row, :], $\mathbf{B}$[next_pivot_row, :])
9:
10:    //Make the leading entry a "1".
11:    $\mathbf{B}$[pivot_row, :] ← $\mathbf{B}$[pivot_row, pivot_column]$^{-1}$$\mathbf{B}$[pivot_row, :]
12:
13:    //Eliminate entries below the pivot.
14:    for row from next_pivot_row $+ 1$ to $m - 1$ do
15:        $\mathbf{B}$[row, :] ← $\mathbf{B}$[row, :] $-$ $\mathbf{B}$[row, pivot_column]$\mathbf{B}$[pivot_row, :]
16:
17:    pivot_row ← pivot_row $+ 1$
18:    pivot_column ← pivot_column $+ 1$
19: return $\mathbf{B}$
```

Algorithm 2 SampleSolution$(\mathbf{A}, \mathbf{y}, \mathbf{r})$

```
Input: Matrix $\mathbf{A} \in \mathbb{F}_q^{m \times ko}$
Require: $ko \geq m$
Input: Target vector $\mathbf{y} \in \mathbb{F}_q^m$
Input: Randomness $\mathbf{r} \in \mathbb{F}_q^{ko}$
Output: Solution $\mathbf{x} \in \mathbb{F}_q^n$ that satisfies $\mathbf{Ax} = \mathbf{y}$ if $\mathrm{rank}(\mathbf{A}) = m$; otherwise, output $\perp$.
 1: //Randomize the system using $\mathbf{r}$.
 2: $\mathbf{x} \leftarrow \mathbf{r} \in \mathbb{F}_q^{ko}$
 3: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{Ar}$
 4:
 5: //Put $(\mathbf{A}\ \mathbf{y})$ in echelon form with leading 1's.
 6: $(\mathbf{A}\ \mathbf{y}) \leftarrow \mathsf{EF}(\mathbf{A}\ \mathbf{y})$
 7:
 8: //Check if $\mathbf{A}$ has rank $m$.
 9: if $\mathbf{A}[m-1, :] = \mathbf{0}_n$ then
10:     return $\perp$
11:
12: //Back-substitution
13: for $r$ from $m-1$ to $0$ do
14:     //Let $c$ be the index of first non-zero element of $A[r, :]$.
15:     $\mathbf{x}_c \leftarrow \mathbf{x}_c + \mathbf{y}_r$
16:     $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{y}_r A[:, c]$
17: return $\mathbf{x}$
```

2.1.3 Hashing and randomness expansion

SHAKE256**.** We use the SHAKE256 extended output function for the purpose of hashing and sampling secret material. We denote by SHAKE256(X, l) the function that takes a byte string $X \in \mathcal{B}^*$ and outputs l bytes of output, as specified in the SHA-3 standard [SHA15].

AES**-**128**-**CTR**-based seed expansion.** MAYO uses an AES-128-CTR-based seed expansion function to generate a large part of the coefficients of the multivariate quadratic map $\mathcal{P}$. We define the function AES-128-CTR(seed, l), which takes a 16-byte seed seed, and produces l bytes of output. The output is the concatenation of the AES-128-CTR encryptions of the blocks $0, 1, \cdots, l/16 - 1$ (the blocks are 16-byte counter values starting with zero, and counting up to $l/16 - 1$), using seed as the key in the AES-128-CTR block cipher [AES01]. The implementation of the AES-128-CTR block cipher does not need to be constant-time or side-channel secure, because the key, the input, and the output are all public.

2.1.4 Data types and conversions

The MAYO protocol specified in this document involves operations using several data types. This section lists the different data types and describes how to convert one data type to another.

2.1.4.1 Field element to nibble: Encode$_{\mathbb{F}_{16}}(a) \in [16]$

We encode a field element $a = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ as a nibble Encode$_{\mathbb{F}_{16}}(a) \in [16]$, whose four bits are (from least significant bit to most significant bit) (a_0, a_1, a_2, a_3).

2.1.4.2 Nibble to field element: Decode$_{\mathbb{F}_{16}}$(nibble)

The operation Decode$_{\mathbb{F}_{16}}$ is the inverse of Encode$_{\mathbb{F}_{16}}$. It takes a nibble as input and outputs the corresponding field element.

2.1.4.3 Vector to byte-string: $\mathsf{Encode}_{vec}(\mathbf{x})$

We encode a vector $\mathbf{x} \in \mathbb{F}_{16}^n$ as a string of $\lceil n/2 \rceil$ bytes by concatenating the encodings of the field elements $\mathsf{Encode}_{\mathbb{F}_{16}}(x_1), \ldots, \mathsf{Encode}_{\mathbb{F}_{16}}(x_n)$, and padding with the zero nibble if n is odd.

2.1.4.4 Byte-string to vector: $\mathsf{Decode}_{vec}(n, \mathsf{bytestring})$

The operation $\mathsf{Decode}_{vec}(n, \mathsf{bytestring})$ takes a vector length n and a byte-string $\mathsf{bytestring} \in \mathcal{B}^{\lceil n/2 \rceil}$ as input, and outputs a vector in $\mathbb{F}_{16}^n$, such that $\mathsf{Decode}_{vec}(n, \mathsf{Encode}_{vec}(\mathbf{x})) = \mathbf{x}$ for all $n \in \mathbb{N}$ and all $\mathbf{x} \in \mathbb{F}_{16}^n$.

2.1.4.5 Matrix to byte-string: $\mathsf{Encode}_{\mathbf{O}}(\mathbf{O})$

We define the encoding function $\mathsf{Encode}_{\mathbf{O}}(\mathbf{O})$ that encodes a matrix $\mathbf{O} \in \mathbb{F}_{16}^{(n-o)\times o}$ in row-major order to a byte-string. More precisely, $\mathsf{Encode}_{\mathbf{O}}$ first concatenates the $n-o$ rows of O to make a single vector $\mathbf{v} = (\mathbf{O}[0,:]\ \mathbf{O}[1,:]\ \ldots\ \mathbf{O}[n-o-1,:])$ of length $(n-o)o$, and then it outputs $\mathsf{Encode}_{vec}(\mathbf{v})$.

2.1.4.6 Byte-string to Matrix: $\mathsf{Decode}_{\mathbf{O}}(\mathsf{bytestring})$

The operation $\mathsf{Decode}_{\mathbf{O}}(\mathsf{bytestring})$ takes a byte-string $\mathsf{bytestring}$ as input and outputs a matrix in $\mathbb{F}_q^{(n-o)\times o}$ such that $\mathsf{Decode}_{\mathbf{O}}(\mathsf{Encode}_{\mathbf{O}}(\mathbf{O})) = \mathbf{O}$ for all matrices $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$.

2.1.4.7 Encoding a sequence of m (upper triangular) matrices.

Algorithm 3 defines an encoding function $\mathsf{EncodeMatrices}$ to encode a sequence of m (possibly upper trianular) matrices $\mathbf{A}_0, \ldots, \mathbf{A}_{m-1} \in \mathbb{F}_q^{r\times c}$.

The function encodes the entries of m-matrices in row-major order, resulting in m sequences of $r \times c$ nibbles ($r(r+1)/2$ nibbles if the matrices are upper-triangular). Then it interleaves the m sequences.

This means that the encoding starts with m nibbles, which correspond to the m top-left entries of the matrices, i.e., $\mathbf{A}_0[0,0], \ldots, \mathbf{A}_{m-1}[0,0]$. These are followed by the $\{\mathbf{A}_i[0,1]\}_{i\in[m]}$ entries etc. up to the $\{\mathbf{A}_i[0,c-1]\}_{i\in[m]}$ entries. Then we continue with the next row starting with $\{\mathbf{A}_i[1,0]\}_{i\in[m]}$ and so on. The last m nibbles of the encoding correspong to $\{\mathbf{A}_i[r-1,c-1]\}_{i\in[m]}$.

If the $\mathbf{A}_i$ matrices are upper triangular, then $\mathsf{EncodeMatrices}$ works in the same way except that it skips all the field elements $\mathbf{A}_k[i,j]$ with $0 \le j < i < r$.

Algorithm 3 $\mathsf{EncodeMatrices}(r, c, \{\mathbf{A}\}_{i\in[m]}, \mathsf{is\_triangular})$

Input: r, c, the number of rows and columns of the matrices
Input: m matrices $\mathbf{A}_i \in \mathbb{F}_{16}^{r\times c}$
Input: $\mathsf{is\_triangular} \in \{0,1\}$, a bit to indicate if the $\mathbf{A}_i$ are upper triangular or not.
Output: A byte string $\mathsf{bytestring} \in \mathcal{B}^{mrc/2}$ if $\mathsf{is\_triangular} = \mathsf{false}$, $\mathsf{bytestring} \in \mathcal{B}^{mr(r+1)/4}$ otherwise.

```
1: bytestring = $\emptyset$
2: for $i$ from 0 to $r-1$ do
3:     for $j$ from 0 to $c-1$ do
4:         if $i \le j$ or is_triangular = false then
5:             bytestring = bytestring||$\mathsf{Encode}_{vec}(\{\mathbf{A}_k[i,j]\}_{k\in[m]})$
6: return bytestring.
```

We define the encoding function for the sequences of matrices $\mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{P}^{(3)}$, and $\mathbf{L}$ as:

1. $\mathsf{Encode}_{\mathbf{P}^{(1)}}(\cdot) := \mathsf{EncodeMatrices}(n-o, n-o, \cdot, \mathsf{true})$
2. $\mathsf{Encode}_{\mathbf{P}^{(2)}}(\cdot) := \mathsf{EncodeMatrices}(n-o, o, \cdot, \mathsf{false})$

3. $\mathsf{Encode}_{\mathbf{P}^{(3)}}(\cdot) := \mathsf{EncodeMatrices}(o, o, \cdot, \mathsf{true})$

4. $\mathsf{Encode}_{\mathbf{L}}(\cdot) := \mathsf{Encode}_{\mathbf{P}^{(2)}}(\cdot)$,

and we define $\mathsf{Decode}_{\mathbf{P}^{(1)}}$, $\mathsf{Decode}_{\mathbf{P}^{(2)}}$, $\mathsf{Decode}_{\mathbf{P}^{(3)}}$, and $\mathsf{Decode}_{\mathbf{L}}$ to be the inverses of $\mathsf{Encode}_{\mathbf{P}^{(1)}}$, $\mathsf{Encode}_{\mathbf{P}^{(2)}}$, $\mathsf{Encode}_{\mathbf{P}^{(3)}}$, and $\mathsf{Encode}_{\mathbf{L}}$, respectively.

2.1.5 The Basic MAYO functionalities

We define five functionalities:

- MAYO.CompactKeyGen (Algorithm 4): outputs a pair $(\mathsf{csk}, \mathsf{cpk}) \in \mathcal{B}^{\mathsf{csk\_bytes}} \times \mathcal{B}^{\mathsf{cpk\_bytes}}$, where csk and cpk are compact representations of a MAYO secret key and public key respectively.
- MAYO.ExpandSK (Algorithm 5): takes as input csk, the compact representation of a MAYO secret key, and outputs $\mathsf{esk} \in \mathcal{B}^{\mathsf{esk\_bytes}}$, an expanded representation of the secret key.
- MAYO.ExpandPK (Algorithm 6): takes as input cpk, the compact representation of a MAYO public key, and outputs $\mathsf{epk} \in \mathcal{B}^{\mathsf{epk\_bytes}}$, an expanded representation of the public key.
- MAYO.Sign (Algorithm 7): takes an expanded secret key esk, a message $\mathsf{M} \in \mathcal{B}^*$, and outputs a signature $\mathsf{sig} \in \mathcal{B}^{\mathsf{sig\_bytes}}$.
- MAYO.Verify (Algorithm 8): takes as input a message M, an expanded public key epk, a signature sig, and salt, and outputs 1 or 0 if the signature is deemed valid or invalid, respectively.

Algorithm 4 MAYO.CompactKeyGen()

Output: Compact representation of a secret key $\mathsf{csk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$
Output: Compact representation of a public key $\mathsf{cpk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$

```
1:  //Pick $\mathsf{seed_{sk}}$ at random.
2:  $\mathsf{seed_{sk}} \xleftarrow{\$} \mathcal{B}^{\mathsf{sk\_seed\_bytes}}$
3:
4:  //Derive $\mathsf{seed_{pk}}$ and $\mathbf{O}$ from $\mathsf{seed_{sk}}$.
5:  $\mathsf{S} \leftarrow \mathsf{SHAKE256}(\mathsf{seed_{sk}}, \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes})$
6:  $\mathsf{seed_{pk}} \leftarrow \mathsf{S}[0 : \mathsf{pk\_seed\_bytes}]$    // $\mathsf{seed_{pk}} \in \mathcal{B}^{\mathsf{pk\_seed\_bytes}}$
7:  $\mathbf{O} \leftarrow \mathsf{Decode}_{\mathbf{O}}(\mathsf{S}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes}])$    // $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$
8:
9:  //Derive the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from $\mathsf{seed_{pk}}$.
10: $\mathsf{P} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed_{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes})$
11: $\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P}[0 : \mathsf{P1\_bytes}])$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
12: $\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P}[\mathsf{P1\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes}])$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
13:
14: //Compute the $\mathbf{P}_i^{(3)}$.
15: for $i$ from $0$ to $m-1$ do
16:     $\mathbf{P}_i^{(3)} \leftarrow \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$    // $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ upper triangular
17:
18: //Encode the $\mathbf{P}_i^{(3)}$.
19: $\mathsf{cpk} \leftarrow \mathsf{seed_{pk}} \,\|\, \mathsf{Encode}_{\mathbf{P}^{(3)}}(\{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
20: $\mathsf{csk} \leftarrow \mathsf{seed_{sk}}$
21:
22: //Output keys.
23: return $(\mathsf{cpk}, \mathsf{csk})$.
```

Remark. *In lines 31,32, and 34 of the MAYO.Sign algorithm and line 25 of the MAYO.Verify algorithm we accumulate values of the form $\mathbf{E}^l\mathbf{y}$, where $\mathbf{E} \in \mathbb{F}_q^{m\times m}$ is a matrix that represent multiplication by z in a finite field*

Algorithm 5 MAYO.ExpandSK(csk)

Input: Compacted secret key $\mathsf{csk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$

Output: Expanded secret key $\mathsf{esk} \in \mathcal{B}^{\mathsf{esk\_bytes}}$

```
1: //Parse csk
2: $\mathsf{seed_{sk}} \leftarrow \mathsf{csk}[0 : \mathsf{sk\_seed\_bytes}]$
3:
4: //Derive $\mathsf{seed_{pk}}$ and $\mathbf{O}$ from $\mathsf{seed_{sk}}$.
5: $\mathsf{S} \leftarrow \mathsf{SHAKE256}(\mathsf{seed_{sk}}, \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes})$
6: $\mathsf{seed_{pk}} \leftarrow \mathsf{S}[0 : \mathsf{pk\_seed\_bytes}]$    // $\mathsf{seed_{pk}} \in \mathcal{B}^{\mathsf{pk\_seed\_bytes}}$
7: $\mathsf{O\_bytestring} \leftarrow \mathsf{S}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes}]$    // $\mathsf{O\_bytestring} \in \mathcal{B}^{\mathsf{O\_bytes}}$
8: $\mathbf{O} \leftarrow \mathsf{Decode}_{\mathbf{O}}(\mathsf{O\_bytestring})$    // $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$
9:
10: //Derive the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from $\mathsf{seed_{pk}}$.
11: $\mathsf{P} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed_{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes})$
12: $\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P}[0 : \mathsf{P1\_bytes}])$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
13: $\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P}[\mathsf{P1\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes}])$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
14:
15: //Compute the $\mathbf{L}_i$.
16: for $i$ from $0$ to $m-1$ do
17:     $\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)}$    // $\mathbf{L}_i \in \mathbb{F}_q^{(n-o)\times o}$
18:
19: //Encode the $\mathbf{L}_i$ and output esk.
20: return $\mathsf{esk} = \mathsf{seed_{sk}} \parallel \mathsf{O\_bytestring} \parallel \mathsf{P}[0 : \mathsf{P1\_bytes}] \parallel \mathsf{Encode}_{\mathbf{L}}(\{\mathbf{L}_i\}_{i\in[m]})$.
```

Algorithm 6 MAYO.ExpandPK(cpk)

Input: Compact public key $\mathsf{cpk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$

Output: Expanded public key $\mathsf{epk} \in \mathcal{B}^{\mathsf{epk\_bytes}}$

```
1: //Parse cpk.
2: $\mathsf{seed_{pk}} \leftarrow \mathsf{cpk}[0 : \mathsf{pk\_seed\_bytes}]$
3:
4: //Expand $\mathsf{seed_{pk}}$ and return.
5: $\mathsf{epk} = \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed_{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes}) \parallel \mathsf{cpk}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{P3\_bytes}]$
6: return epk.
```

Algorithm 7 MAYO.Sign(esk, M)

Input: Expanded secret key esk $\in \mathcal{B}^{\text{esk\_bytes}}$
Input: Message M $\in \mathcal{B}^*$
Constant: $\mathbf{E} \in \mathbb{F}_q^{m \times m}$ // Represents multiplication by z in $\mathbb{F}_q[z]/(f(z))$
Output: Signature sig $\in \mathcal{B}^{\text{sig\_bytes}}$

```
1: //Decode esk.
2: seed_sk ← esk[0 : sk_seed_bytes]
3: $\mathbf{O}$ ← Decode_O(esk[sk_seed_bytes : sk_seed_bytes + O_bytes])
4: $\{\mathbf{P}_i^{(1)}\}_{i \in m}$ ← Decode_P(1)(esk[sk_seed_bytes + O_bytes : sk_seed_bytes + O_bytes + P1_bytes])
5: $\{\mathbf{L}_i\}_{i \in m}$ ← Decode_L(esk[sk_seed_bytes + O_bytes + P1_bytes : esk_bytes])    // $\mathbf{L}_i \in \mathbb{F}_q^{(n-o) \times o}$
6:
7: //Hash message, and derive salt and $\mathbf{t}$.
8: M_digest ← SHAKE256($M$, digest_bytes)    // M_digest $\in \mathcal{B}^{\text{digest\_bytes}}$
9: R ← $0_{\text{R\_bytes}}$ or R $\xleftarrow{\$}$ $\mathcal{B}^{\text{R\_bytes}}$    // Optional randomization
10: salt ← SHAKE256(M_digest ‖ R ‖ seed_sk, salt_bytes)    // salt $\in \mathcal{B}^{\text{salt\_bytes}}$
11: $\mathbf{t}$ ← Decode_vec($m$, SHAKE256(M_digest ‖ salt, $\lceil m \log(q)/8 \rceil$))    // $\mathbf{t} \in \mathbb{F}_q^m$
12:
13: //Attempt to find a preimage for $\mathbf{t}$.
14: for ctr from 0 to 255 do
15:     //Derive $\mathbf{v}_i$ and $\mathbf{r}$.
16:     V ← SHAKE256(M_digest ‖ salt ‖ seed_sk ‖ ctr, $k *$ v_bytes + $\lceil ko \log(q)/8 \rceil$)
17:     for $i$ from 0 to $k-1$ do
18:         $\mathbf{v}_i$ ← Decode_vec($n-o$, V[$i *$ v_bytes : $(i+1) *$ v_bytes])    // $\mathbf{v}_i \in \mathbb{F}_q^{n-o}$
19:     $\mathbf{r}$ ← Decode_vec($ko$, V[$k *$ v_bytes : $k *$ v_bytes + $\lceil ko \log(q)/8 \rceil$])
20:
21:     //Build linear system $\mathbf{Ax} = \mathbf{y}$.
22:     $\mathbf{A} \leftarrow 0_{m \times ko} \in \mathbb{F}_q^{m \times ko}$
23:     $\mathbf{y} \leftarrow \mathbf{t}, \ell \leftarrow 0$
24:     for $i$ from 0 to $k-1$ do
25:         $\mathbf{M}_i \leftarrow \mathbf{0}_{m \times o} \in \mathbb{F}_q^{m \times o}$
26:         for $j$ from 0 to $m-1$ do
27:             $\mathbf{M}_i[j,:] \leftarrow \mathbf{v}_i^{\mathsf{T}} \mathbf{L}_j$    // Set $j$-th row of $\mathbf{M}_i$
28:     for $i$ from 0 to $k-1$ do
29:         for $j$ from $k-1$ to $i$ do
30:             $\mathbf{u} = \begin{cases} \{\mathbf{v}_i^{\mathsf{T}} \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a \in [m]} & \text{if } i = j \\ \{\mathbf{v}_i^{\mathsf{T}} \mathbf{P}_a^{(1)} \mathbf{v}_j + \mathbf{v}_j^{\mathsf{T}} \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a \in [m]} & \text{if } i \neq j \end{cases}$    // $\mathbf{u} \in \mathbb{F}_q^m$
31:             $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}^\ell \mathbf{u}$
32:             $\mathbf{A}[:, i*o : (i+1)*o] \leftarrow \mathbf{A}[:, i*o : (i+1)*o] + \mathbf{E}^\ell \mathbf{M}_j$
33:             if $i \neq j$ then
34:                 $\mathbf{A}[:, j*o : (j+1)*o] \leftarrow \mathbf{A}[:, j*o : (j+1)*o] + \mathbf{E}^\ell \mathbf{M}_i$
35:             $\ell \leftarrow \ell + 1$
36:
37:     //Try to solve the system.
38:     $\mathbf{x}$ ← SampleSolution($\mathbf{A}, \mathbf{y}, \mathbf{r}$)    // $\mathbf{x} \in \mathbb{F}_q^{ko} \cup \{\perp\}$
39:     if $\mathbf{x} \neq \perp$ then
40:         break
41:
42: //Finish and output the signature.
43: $\mathbf{s} \leftarrow \mathbf{0}_{kn}$    // $\mathbf{s} \in \mathbb{F}_q^{kn}$
44: for $i$ from 0 to $k-1$ do
45:     $\mathbf{s}[i*n : (i+1)*n] \leftarrow (\mathbf{v}_i + \mathbf{Ox}[i*o : (i+1)*o]) \parallel \mathbf{x}[i*o : (i+1)*o]$
46: return sig = Encode_vec($\mathbf{s}$) ‖ salt.
```

Algorithm 8 MAYO.Verify(epk, M, sig)

Input: Expanded public key epk $\in \mathcal{B}^{\text{epk\_bytes}}$

Input: Message M $\in \mathcal{B}^*$

Input: Signature sig $\in \mathcal{B}^{\text{sig\_bytes}}$

Constant: $\mathbf{E} \in \mathbb{F}_q^{m \times m}$ // Represents multiplication by z in $\mathbb{F}_q[z]/(f(z))$

Output: An integer result to indicate if sig is valid (result $= 0$) or invalid (result < 0).

```
1:  //Decode epk.
2:  P1_bytestring ← epk[0 : P1_bytes]
3:  P2_bytestring ← epk[P1_bytes : P1_bytes + P2_bytes]
4:  P3_bytestring ← epk[P1_bytes + P2_bytes : P1_bytes + P2_bytes + P3_bytes]
5:  $\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P1\_bytestring})$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
6:  $\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P2\_bytestring})$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
7:  $\{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(3)}}(\mathsf{P3\_bytestring})$    // $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ upper triangular
8:
9:  //Decode sig.
10: salt ← sig[$\lceil nk/2 \rceil$ : $\lceil nk/2 \rceil$ + salt_bytes]
11: $\mathbf{s} \leftarrow \mathsf{Decode}_{vec}(kn, \mathsf{sig})$
12: for $i$ from 0 to $k-1$ do
13:     $\mathbf{s}_i \leftarrow \mathbf{s}[i*n : (i+1)*n]$
14:
15: //Hash message and derive $\mathbf{t}$.
16: M_digest ← SHAKE256(M, digest_bytes)    // M_digest $\in \mathcal{B}^{\text{digest\_bytes}}$
17: $\mathbf{t} \leftarrow \mathsf{Decode}_{vec}(m, \mathsf{SHAKE256}(\mathsf{M\_digest} \parallel \mathsf{salt}, \lceil m\log(q)/8 \rceil))$    // $\mathbf{t} \in \mathbb{F}_q^m$
18:
19: //Compute $\mathcal{P}^*(\mathbf{s})$.
20: $\mathbf{y} \leftarrow \mathbf{0}_m$    // $\mathbf{y} \in \mathbb{F}_q^m$
21: $\ell \leftarrow 0$
22: for $i$ from 0 to $k-1$ do
23:     for $j$ from $k-1$ to $i$ do
24:         $\mathbf{u} \leftarrow \begin{cases} \left\{ \mathbf{s}_i^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i = j \\ \left\{ \mathbf{s}_i^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_j + \mathbf{s}_j^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i \neq j \end{cases}$    // $\mathbf{u} \in \mathbb{F}_q^m$
25:         $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}^\ell \mathbf{u}$
26:         $\ell \leftarrow \ell + 1$
27:
28: //Accept signature if $\mathbf{y} = \mathbf{t}$.
29: if $\mathbf{y} = \mathbf{t}$ then
30:     return 0
31: return $-1$
```

$\mathbb{F}_q[z]/f(z)$. *Rather than computing the matrix multiplications explicitly, it could be more efficient to accumulate the values in a single polynomial and do a single reduction mod* $f(z)$.

Remark. *Line 24 of the MAYO.Verify algorithm repeatedly uses the values* $\mathbf{s}_i^\mathsf{T}\begin{pmatrix}\mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)}\end{pmatrix}$. *To get an efficient implementation, these values can be computed only once and reused, as opposed to recomputing them in every iteration of the for-loop. The same holds for the values* $\mathbf{v}_i^\mathsf{T}\mathbf{P}_a^{(1)}$ *on line 30 of MAYO.Sign.*

2.1.6 Implementing the NIST API

Using the five basic functionalities, we implement the NIST API with the following three algorithms:

- MAYO.API.keypair (Algorithm 4): Outputs a pair $(\mathsf{sk}, \mathsf{pk}) \in \mathcal{B}^{\mathsf{csk\_bytes}} \times \mathcal{B}^{\mathsf{cpk\_bytes}}$, where sk and pk are compact representations of a MAYO secret key and public key respectively. This algorithm is identical to MAYO.CompactKeyGen.
- MAYO.API.sign (Algorithm 9): Takes as input a secret key $\mathsf{sk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$ and a message $\mathsf{M} \in \mathcal{B}^{\mathsf{mlen}}$. It first calls MAYO.ExpandSK to expand the secret key, and then calls MAYO.Sign with the expanded public key to produce the signature. It outputs a signed message $\mathsf{sm} \in \mathcal{B}^{\mathsf{sig\_bytes}+\mathsf{mlen}}$, which is the concatenation of the signature and the message.
- MAYO.API.sign_open (Algorithm 10): Takes as input a signed message $\mathsf{sm} \in \mathcal{B}^*$ and a public key $\mathsf{pk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$. It first calls MAYO.ExpandPK to expand the public key, and then it calls MAYO.Verify with the expanded public key to check if the signature is valid. It outputs the result of MAYO.Verify and if the signature is valid it also outputs the original message.

Algorithm 9 MAYO.API.sign(M, sk)

Input: Secret key $\mathsf{sk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$
Input: A message $\mathsf{M} \in \mathcal{B}^{\mathsf{mlen}}$
Output: A signed message $\mathsf{sm} \in \mathcal{B}^{\mathsf{sig\_bytes}+\mathsf{mlen}}$.

```
1: //Expand sk.
2: esk ← MAYO.ExpandSK(pk)
3:
4: //Produce signature.
5: sig ← MAYO.Sign(esk, M)
6:
7: //Return signed message.
8: return sig || M
```

Algorithm 10 MAYO.API.sign_open(pk, M, sig)

```
Input: Public key pk $\in \mathcal{B}^{\mathsf{cpk\_bytes}}$
Input: Signed message sm $\in \mathcal{B}^{\mathsf{smlen}}$
Output: An integer result to indicate if sm is valid (result $= 0$) of invalid (result $< 0$) for pk
Output: The original message M $\in \mathcal{B}^{\mathsf{smlen}-\mathsf{sig\_bytes}}$ if sm is valid.
1: //Expand pk.
2: epk ← MAYO.ExpandPK(pk)
3:
4: //Parse signed message.
5: sig ← sm[0 : sig_bytes]
6: M ← sm[sig_bytes : smlen]
7:
8: //Verify signature.
9: result ← MAYO.Verify(epk, $M$, sig)
10:
11: //Return result and message.
12: if result $< 0$ then
13:     M ← ⊥
14: return (result, M)
```

2.1.7 Parameter sets

2.1.7.1 Chosen parameter sets.

We select and implement four parameter sets: For **NIST security level 1**, we select two parameter sets: MAYO_1 and MAYO_2, where MAYO_1 has smaller public keys but larger signatures and conversely MAYO_2 has smaller signatures but larger public keys. For **NIST security level 3** and **NIST security level 5**, we select one parameter set each, which we refer to as MAYO_3 and MAYO_5, respectively. The parameter sets and the corresponding key and signature sizes are displayed in Table 2.1.

Our chosen parameter sets use the following four irreducible polynomials in $\mathbb{F}_{16}[z]$:

$$\begin{array}{lclllll}
f_{64}(z) & = & z^{64} & +x^3z^3 & +xz^2 & & +x^3 \\
f_{78}(z) & = & z^{78} & & +z^2 & +z & +x^3 \\
f_{108}(z) & = & z^{108} & +(x^2+x+1)z^3 & +z^2 & & +x^3 \\
f_{142}(z) & = & z^{142} & +z^3 & +x^3z^2 & & +x^2
\end{array}$$

2.1.7.2 Additional parameter sets.

Table 2.2 gives some additional parameters for NIST security levels 1, 3, and 5 to showcase the possible trade-offs between public key size and signature size for the MAYO signature scheme. Implementations of MAYO with these parameter sets can be obtained with minimal effort by changing parameter values in our implementations of MAYO_1, MAYO_2, MAYO_3, and MAYO_5. The main parameter sets MAYO_1, MAYO_2, MAYO_3, and MAYO_5 are shown in **boldface**.

Table 2.1: Our selection of parameter sets for MAYO. All sizes are reported in bytes (B) or kilobytes (KB).

Parameter set security level	MAYO_1 1	MAYO_2 1	MAYO_3 3	MAYO_5 5
n	86	81	118	154
m	78	64	108	142
o	8	17	10	12
k	10	4	11	12
q	16	16	16	16
salt_bytes	24	24	32	40
digest_bytes	32	32	48	64
pk_seed_bytes	16	16	16	16
$f(z)$	$f_{78}(z)$	$f_{64}(z)$	$f_{108}(z)$	$f_{142}(z)$
secret key size	24 B	24 B	32 B	40 B
public key size	1420 B	4912 B	2986 B	5554 B
signature size	454 B	186 B	681 B	964 B
expanded sk size	141 KB	99 KB	367 KB	822 KB
expanded pk size	142 KB	104 KB	370 KB	828 KB

Table 2.2: Additional parameter sets for MAYO. The salt_bytes, pk_seed_bytes, and digest_bytes parameters are the same as those for the main parameter sets for the same security levels respectively. This implies that the secret key size is 24, 32, or 40 bytes for parameter sets targetting security levels 1,3, and 5 respectively. All sizes are reported in bytes (B) or kilobytes (KB).

Security level	Parameter set (n, m, o, k, q)	pk size	sig size	esk size	epk size
1	**(86, 78, 8, 10, 16)**	**1420 B**	**454 B**	**142 KB**	**143 KB**
	(85, 76, 9, 9, 16)	1726 B	406 B	134 KB	136 KB
	(84, 74, 10, 8, 16)	2051 B	360 B	128 KB	129 KB
	(81, 70, 11, 7, 16)	2326 B	307 B	112 KB	114 KB
	(80, 68, 12, 6, 16)	2668 B	264 B	105 KB	108 KB
	(80, 66, 14, 5, 16)	3481 B	224 B	102 KB	105 KB
	(81, 64, 17, 4, 16)	**4912 B**	**186 B**	**99 KB**	**104 KB**
	(86, 64, 22, 3, 16)	8112 B	153 B	109 KB	117 KB
	(100, 64, 33, 2, 16)	17968 B	124 B	141 KB	158 KB
3	(123,114, 9, 13, 16)	2581 B	831 B	422 KB	425 KB
	(118,108, 10, 11, 16)	**2986 B**	**681 B**	**368 KB**	**371 KB**
	(117,106, 11, 10, 16)	3514 B	617 B	354 KB	358 KB
	(116,104, 12, 9, 16)	4072 B	554 B	341 KB	345 KB
	(115,102, 13, 8, 16)	4657 B	492 B	328 KB	333 KB
	(115,100, 15, 7, 16)	6016 B	434 B	320 KB	326 KB
	(115, 98, 17, 6, 16)	7513 B	377 B	312 KB	320 KB
	(118, 98, 20, 5, 16)	10306 B	327 B	326 KB	336 KB
	(121, 96, 25, 4, 16)	15616 B	274 B	331 KB	346 KB
	(129, 96, 33, 3, 16)	26944 B	225 B	367 KB	394 KB
	(150, 96, 49, 2, 16)	58816 B	182 B	474 KB	531 KB
5	(156,146, 10, 15, 16)	4031 B	1210 B	870 KB	874 KB
	(155,144, 11, 14, 16)	4768 B	1125 B	846 KB	851 KB
	(154,142, 12, 12, 16)	**5554 B**	**964 B**	**823 KB**	**828 KB**
	(153,140, 13, 11, 16)	6386 B	881 B	800 KB	806 KB
	(152,138, 14, 10, 16)	7261 B	800 B	777 KB	784 KB
	(152,136, 16, 9, 16)	9264 B	724 B	764 KB	773 KB
	(151,134, 17, 8, 16)	10267 B	644 B	741 KB	751 KB
	(152,132, 20, 7, 16)	13876 B	572 B	736 KB	750 KB
	(155,132, 23, 6, 16)	18232 B	505 B	762 KB	780 KB
	(157,130, 27, 5, 16)	24586 B	432 B	764 KB	788 KB
	(161,128, 33, 4, 16)	35920 B	362 B	780 KB	816 KB
	(172,128, 44, 3, 16)	63376 B	298 B	868 KB	930 KB
	(202,128, 65, 2, 16)	137296 B	242 B	1148 KB	1282 KB

Chapter 3

Detailed performance analysis

The submission package includes:

1. A generic reference implementation written only in portable C (C99), described in Section 3.1.
2. An optimized implementation written only in portable C (C99), described in Section 3.2.
3. An additional, Intel AVX2 optimized implementation written in C (C99) and using assembly compiler intrinsics, described in Section 3.3.
4. An additional, Arm NEON optimized implementation written in C (C99) and using assembler compiler intrinsics, as described in Section 3.4.
5. An additional, Arm Cortex-M4 optimized implementation written in C (C99) and assembly code, as described in Section 3.5.
6. An additional, simple textbook implementation written exclusively in Sage, described in Section 3.6.

The implementations 1-4 are delivered in a common code package, where each implementation can be compiled and built by providing the respective cmake options. For portability purposes, the code package does not make use of dynamic memory allocation or variable length arrays. The Cortex-M4 optimized implementation is delivered as a separate code package. Libraries supporting the NIST Signature API are built for each parameter set, along with a test harness to verify the Known Answer Tests (KAT) and applications to generate the KAT. For a detailed overview of the build options and built artifacts, we refer to the "README.md" file in the source code package submitted along with the specification.

All implementations except the Sage textbook implementation are protected against side-channel attacks on the software level: they avoid secret-dependent data indexing and secret-dependent control flow.

3.1 Reference implementation

The reference implementation uses generic functions applicable for all parameter sets, which allows to build the implementation in a single library supporting all parameter sets at run-time. This option leads to a smaller library size and makes it easier for the consumer to use the different MAYO variants in a single library.

The reference implementation is built with CMake option `-DMAYO_BUILD_TYPE=ref` . It is found at `https://github.com/PQCMayo/MAYO-C`.

3.2 Optimized implementation

The optimized C implementation differs in two points from the reference implementation. First, the MAYO parameters are set at compile-time, resulting in separate libraries for each parameter set. Modern compilers are highly able to efficiently unroll matrix arithmetic operations which leads to being able to avoid manually unrolling the loops. Second, specialized batched arithmetic functions are implemented.

A big part of the key expansion computational time is dominated by AES, which allows us to significantly speed up the performance by using an AES implementation that uses AES-NI. However, this speedup may differ depending on the AES software implementation used and the Intel CPU generation.

The optimized implementation is built with CMake option `-DMAYO_BUILD_TYPE=opt`. AES-NI is used by default, if available. It is found at https://github.com/PQCMayo/MAYO-C.

3.3 Intel AVX2 optimized implementation

The AVX2 implementation targets Intel Haswell architectures or later. The implementation utilizes compiler assembly intrinsics for the SSE2, SSSE3, AVX and AVX2 instruction sets. The implementation uses AVX2 shuffle instructions to implement $\mathbb{F}_{16}$ arithmetic on field elements in nibble-sliced encoding, which follows the techniques used in [BCC+24].

The AVX2 implementation is built with CMake option `-DMAYO_BUILD_TYPE=avx2`. AES-NI is used by default, if available. It is found at https://github.com/PQCMayo/MAYO-C.

3.3.1 Performance evaluation on Intel x86-64

We ran the performance evaluation procedure on Intel x86-64 CPU's of three architectures: Haswell, Skylake, and Ice Lake. The library was compiled with the following CMake compile options:

- Reference implementation: `-DMAYO_BUILD_TYPE=ref -DENABLE_AESNI=OFF`
- Optimized implementation (using AES-NI): `-DMAYO_BUILD_TYPE=opt -DENABLE_AESNI=ON`
- Optimized implementation (without AES-NI): `-DMAYO_BUILD_TYPE=opt -DENABLE_AESNI=OFF`
- AVX2 implementation (using AES-NI): `-DMAYO_BUILD_TYPE=avx2 -DENABLE_AESNI=ON`

All builds use `-O3` compiler optimization level and `-march=native` build architecture. Turbo Boost was deactivated to achieve consistent timings. In Tables 3.1, 3.2, and 3.3, we list the performance using the four configurations. We see that the use of AES-NI significantly speeds up the overall performance in operations using key expansion.

Using AVX2 optimizations lead to a speed-up factor between approx. $4\times$-$12\times$ in KeyGen, $5\times$-$9\times$ in Signing (+ExpandSK) and $1.6\times$-$2.5\times$ in Verifying (+ExpandPK).

The fastest results on Intel x86-46 are on the 2.0 GHz Ice Lake platform on which MAYO$_2$ performs KeyGen in 48×10^{-6} s, Signing (+ExpandSK) in 143×10^{-6} s, and Verifying (+ExpandPK) in 28×10^{-6} s. Batch signing (without expandSK) takes 89×10^{-6} s, and batch verification (without expandPK) takes 11×10^{-6} s.

3.4 Arm NEON implementation

The NEON implementation is built with CMake option `-DMAYO_BUILD_TYPE=neon`. AES acceleration is used by default, if available. We have tested the implementation on Apple M1, M2, and M3 processors.

Table 3.1: MAYO performance in CPU cycles on an Intel Xeon E3-1225 v3 CPU (**Haswell**) at 3.20GHz. The library was compiled on Ubuntu with clang version 18.1.3. Results are the median of 1000 benchmark runs.

Scheme	`KeyGen`	`ExpandSK`	`ExpandPK`	`ExpandSK` + `Sign`	`ExpandPK` + `Verify`
Reference Implementation			Generic portable C code, no AES-NI		
MAYO$_1$	6,136,664	6,718,475	3,627,190	10,328,148	4,709,958
MAYO$_2$	6,348,792	6,118,928	2,520,575	7,272,585	2,995,910
MAYO$_3$	17,263,145	19,624,527	9,386,543	29,259,792	11,669,640
MAYO$_5$	418,26,120	47,847,047	20,930,209	69,377,987	24,046,202
Optimized Implementation		C code, using AES-NI (1st row), no AES-NI (2nd row)			
MAYO$_1$	1,052,858	1,742,144	164,227	3,171,356	530,101
	4,495,455	5,167,549	3,590,318	6,607,738	3,971,862
MAYO$_2$	1,275,893	1,880,686	69,568	2,422,209	148,387
	3,683,763	4,289,329	2,469,638	4,826,897	2,554,868
MAYO$_3$	3,230,269	7,090,999	398,097	10,725,576	1,313,842
	12,164,881	16,004,864	9,324,857	19,651,290	10,239,800
MAYO$_5$	8,041,769	14,006,088	802,679	21,918,131	2,400,637
	27,980,233	33,990,628	20,734,159	41,824,890	22,360,915
AVX2 Optimized Implementation			AVX2 compiler intrinsics and using AES-NI		
MAYO$_1$	246,458	289,936	164,286	702,261	290,041
MAYO$_2$	153,420	168,781	69,468	375,493	96,606
MAYO$_3$	574,472	690,839	399,241	1,476,585	664,631
MAYO$_5$	1,338,275	1,823,962	802,931	3,475,547	1,488,808

Table 3.2: MAYO performance in CPU cycles on an Intel Xeon E3-1260L v5 CPU (**Skylake**) at 2.90GHz. The library was compiled on Ubuntu with clang version 14.0.0-1ubuntu1 20.04.5. Results are the median of 1000 benchmark runs.

Scheme	`KeyGen`	`ExpandSK`	`ExpandPK`	`ExpandSK` + `Sign`	`ExpandPK` + `Verify`
Reference Implementation			Generic portable C code, no AES-NI		
MAYO_1	4,428,410	5,177,975	3,308,053	7,417,216	4,565,177
MAYO_2	3,776,239	4,297,337	2,298,707	4,913,825	2,803,142
MAYO_3	12,623,942	15,358,373	8,500,103	21,588,512	11,057,587
MAYO_5	27,359,756	33,603,963	19,089,884	44,790,568	24,618,130
Optimized Implementation		C code, using AES-NI (1st row), no AES-NI (2nd row)			
MAYO_1	899,214	1,389,824	105,474	2,797,221	441,935
	3,997,247	4,489,772	3,222,065	5,908,751	3,555,313
MAYO_2	1,103,437	1,437,613	64,316	1,868,567	142,224
	3,315,739	3,613,649	2,238,274	4,041,581	2,314,740
MAYO_3	2,998,299	4,930,627	277,937	8,615,942	1,119,046
	1,1070,865	13,049,336	8,389,344	16,697,238	9,197,004
MAYO_5	8,372,676	13,663,263	618,355	23,777,500	2,106,202
	26,417,892	36,818,182	18,670,437	41,815,053	20,261,216
AVX2 Optimized Implementation			AVX2 compiler intrinsics and using AES-NI		
MAYO_1	202,026	236,448	105,454	574,530	247,169
MAYO_2	157,929	169,402	64,231	327,303	96,266
MAYO_3	473,424	604,958	277,497	1,260,260	589,375
MAYO_5	1,197,019	1,707,491	617,173	3,037,778	11,336,685

Table 3.3: MAYO performance in CPU cycles on an Intel Xeon Gold 6338 CPU (**Ice Lake**) with 2.0 GHz. The library was compiled with Ubuntu clang version 18.1.8. Results are the median of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
Reference Implementation			Generic portable C code, no AES-NI		
MAYO$_1$	5,003,442	5,959,158	3,090,332	9,012,970	4,146,684
MAYO$_2$	4,675,660	5,424,294	2,152,894	6,328,160	2,658,042
MAYO$_3$	13,491,044	20,241,010	7,971,570	25,371,070	10,214,798
MAYO$_5$	32,066,684	42,532,676	17,802,768	60,742,348	21,607,318
Optimized Implementation			C code, using AES-NI (1st row), no AES-NI (2nd row)		
MAYO$_1$	878,654	968,112	59,168	2,724,374	353,682
	3,826,622	3,948,928	3,022,428	5,677,050	3,337,256
MAYO$_2$	775,382	1,128,820	33,380	1,541,636	104,290
	2,828,124	3,217,220	2,116,456	3,626,054	2,158,068
MAYO$_3$	3,485,616	3,335,336	151,642	8,009,012	877,112
	11,223,550	11,092,642	7,850,506	15,680,444	8,569,854
MAYO$_5$	7,063,608	11,878,074	339,616	20,790,296	1,775,574
	24,308,544	29,294,506	17,573,154	37,976,996	18,982,458
AVX2 Optimized Implementation			AVX2 compiler intrinsics and using AES-NI		
MAYO$_1$	118,704	162,746	59,254	471,028	153,266
MAYO$_2$	96,288	108,542	33,536	286,028	56,374
MAYO$_3$	282,446	403,716	151,636	1,017,216	347,972
MAYO$_5$	766,682	1,185,704	341,500	2,387,350	853,920

Table 3.4: MAYO performance of the NEON optimized implementation in CPU cycles on an Apple M1 Max. The library was compiled with the Apple clang toolchain (`clang-1600.0.26.3`). Results are the average of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
MAYO_1	133,363	163,985	75,228	434,861	168,769
MAYO_2	128,860	132,219	47,015	292,570	73,303
MAYO_3	380,032	505,894	199,167	1,073,705	467,230
MAYO_5	897,532	1,187,439	446,215	2,278,066	967,922

Table 3.5: MAYO performance of the NEON optimized implementation in CPU cycles on an Apple M3. The library was compiled with the Apple clang toolchain (`clang-1600.0.26.6`). Results are the average of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
MAYO_1	131,133	160,974	73,479	429,283	169,516
MAYO_2	125,437	129,119	44,679	279,380	72,482
MAYO_3	386,820	490,759	194,321	1,039,799	454,799
MAYO_5	895,085	1,158,561	432,300	2,217,257	948,399

The implementation is found at `https://github.com/PQCMayo/MAYO-C`. Table 3.4 and Table 3.5 shows benchmark results for M1 and M3, respectively.

3.5 Arm Cortex-M4 implementation

Our Arm Cortex-M4 implementation is available at `https://github.com/PQCMayo/MAYO-M4`.

For benchmarking, we use the ST NUCLEO-L4R5ZI development board which features a STM32L4R5ZI Cortex-M4 CPU with 2 MB of flash memory and 640 KB of SRAM. We make use of the benchmarking framework PQM4 [KPR+] and use their implementations of Keccak and AES, i.e., we use the `Armv7-M` Keccak implementation from the XKCP [DHP+] and the T-table AES implementation of Stoffelen and Schwabe [SS16]. Note that AES is only used for expanding public values and, hence, using the T-table implementation (instead of the slower constant-time bit-sliced implementation) is acceptable even on Arm Cortex-M4 platforms with a data cache. All implementations were compiled with `-O3` using the Arm GNU toolchain (`arm-none-eabi-gcc`, version 13.3.1).

Table 3.6: MAYO performance in CPU cycles on the **Arm Cortex-M4** (STM32L4R5ZI). The library was compiled with the Arm GNU toolchain (`arm-none-eabi-gcc` 13.3.1). Results are the average of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
MAYO_1	9,572,510	9,696,399	6,946,209	16,911,188	11,242,036
MAYO_2	9,574,993	8,195,117	4,837,557	11,003,859	6,027,831
MAYO_3	27,170,014	29,678,787	18,077,913	45,428,633	28,446,049

3.6 Sage textbook implementation

The Sage textbook implementation is provided as an easy way to understand the scheme, and to test the KAT values generated by the C code. It is not protected against side-channel attacks on the software level, and should only be used as a reference. It is found at `https://github.com/PQCMayo/MAYO-sage`.

Chapter 4

Known Answer Test values

The submission includes KAT files that contain tuples of secret keys (sk), public keys (pk), signatures (sm), messages (msg), and seeds (seed) for our implementations of MAYO_1, MAYO_2, MAYO_3, and MAYO_5.

The KAT files can be found in the media folder of the submission: **KAT**.

Chapter 5

Security Analysis

This chapter is largely based on the security analysis of [Beu22], but is slightly more detailed. We define two hardness assumptions based on which we tightly prove the MAYO signature scheme to be EUF-CMA secure in the random oracle model (ROM). Since one of the assumptions is relatively new, the security reduction in this chapter does not provide a hard guarantee for the security of the scheme by itself. Still, we hope the security reduction is valuable for cryptanalysts to understand what is necessary to attack our scheme.

5.1 Hard Problems underlying MAYO

The first assumption that we use underlies the security of the *Oil and Vinegar* signature scheme.

Definition 1 (OV problem)**.** *For $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$, let $\mathsf{MQ}_{n,m,q}(\mathbf{O})$ denote the set of multivariate maps $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$ that vanish on the rowspace of $\begin{pmatrix}\mathbf{O}^\mathsf{T} & \mathbf{I}_o\end{pmatrix}$. The OV problem asks to distinguish a random multivariate quadratic map $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$ from a random multivariate quadratic map in $\mathsf{MQ}_{n,m,q}(\mathbf{O})$ for a random $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$.*

Let $\mathcal{A}$ be an OV distinguisher algorithm. We say the distinguishing advantage of $\mathcal{A}$ is:

$$\mathsf{Adv}^{\mathsf{OV}}_{n,m,o,q}(\mathcal{A}) = \left| \Pr\left[\mathcal{A}(\mathcal{P}) = 1 \middle| \mathcal{P} \leftarrow \mathsf{MQ}_{m,n,q}\right] - \Pr\left[\mathcal{A}(\mathcal{P}) = 1 \middle| \begin{array}{c}\mathbf{O} \leftarrow \mathbb{F}_q^{(n-o)\times o} \\ \mathcal{P} \leftarrow \mathsf{MQ}_{n,m,q}(\mathbf{O})\end{array}\right] \right| .$$

The OV problem has been studied since the invention of the *Oil and Vinegar* signature scheme in 1997 and seems relatively well understood.

Our second hardness assumption is tailored to the MAYO signature scheme and is, therefore, a more recent assumption. This assumption states that picking a random multivariate quadratic map $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$ and whipping it up to a larger map $\mathcal{P}^\star \in \mathsf{MQ}_{kn,m,q}$ results in a multi-target preimage resistant function on average.

Definition 2 (Multi-Target Whipped MQ problem)**.** *For some matrices $\{\mathbf{E}_{ij}\}_{1\leq i\leq j\leq k} \in \mathbb{F}_{q^m}$, given random $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$ and access to an unbounded number of random targets $\mathbf{t}_i \in \mathbb{F}_q^m$ for $i \in \mathbb{N}$, the multi-target whipped MQ problem asks to compute $(I, \mathbf{s}_1, \ldots, \mathbf{s}_k)$, such that*

$$\sum_{i=1}^{k} \mathbf{E}_{ii}\mathcal{P}(\mathbf{s}_i) + \sum_{1\leq i<j\leq k} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{s}_i, \mathbf{s}_j) = \mathbf{t}_I \,.$$

Let $\mathcal{A}$ be an adversary. We say that the advantage of $\mathcal{A}$ against the multi-target whipped MQ problem is:

$$\mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{A}) = \Pr\left[\sum_{i=1}^{k} \mathbf{E}_{ii}\mathcal{P}(\mathbf{s}_i) + \sum_{i<j} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{s}_i,\mathbf{s}_j) = \mathbf{t}_I \;\middle|\; \begin{array}{c} \mathcal{P} \leftarrow \mathsf{MQ}_{n,m,q} \\ \{\mathbf{t}_i\} \leftarrow \mathbb{F}_q^{m\times\mathbb{N}} \\ (I,\mathbf{s}_1,\dots,\mathbf{s}_k) \leftarrow \mathcal{A}^{\mathbf{t}_i}(\mathcal{P}) \end{array}\right].$$

5.2 Security Proof

In this section, we prove the following theorem:

Theorem 1. *Let $\mathcal{A}$ be an* EUF-CMA *adversary that runs in time T against the MAYO signature in the random oracle model with parameters as in Section 2.1.1, and which makes at most Q_s signing queries and at most Q_h queries to the random oracle. Let* $\mathsf{B} = \frac{q^{k-(n-o)}}{q-1} + \frac{q^{m-ko}}{q-1}$ *be the bound on the failing probability from Lemma 1 and suppose $Q_s\mathsf{B} < 1$, then there exist adversaries $\mathcal{B}^{\mathcal{A}}$ and $\mathcal{B}'^{\mathcal{A}}$ against the* $\mathsf{OV}_{n,m,o,q}$ *and* $\mathsf{MTWMQ}_{\{\mathbf{E}_{ij}\},n,m,k,q}$ *problems respectively, that run in time $T + (Q_s + Q_h + 1)\cdot poly(n,m,k,q)$ such that:*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{n,m,o,k,q}(\mathcal{A}) \le \left(\mathsf{Adv}^{\mathsf{OV}}_{n,m,o,q}(\mathcal{B}^{\mathcal{A}}) + \mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{B}'^{\mathcal{A}})\right)(1-Q_s\mathsf{B})^{-1}$$
$$+ (Q_h+Q_s)Q_s 2^{-8\mathsf{salt\_bytes}} + 3Q_h 2^{-8\mathsf{sk\_seed\_bytes}} + (Q_s+Q_h+2)^2 2^{-8\mathsf{digest\_bytes}}.$$

Before we give the proof, which is an adaptation of the proof strategy for PSS [BR98], we recall a lemma from [Beu22], that gives an upper bound for the probability that the signing algorithm needs to restart because the matrix $\mathbf{A}$ does not have rank m.

Lemma 1. *For $0 \le i \le j < k$, let the $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ be matrices such that*

$$\mathbf{E} = \begin{pmatrix} \mathbf{E}_{11} & \mathbf{E}_{12} & \dots & \mathbf{E}_{1k} \\ \mathbf{E}_{12} & \mathbf{E}_{22} & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{E}_{1k} & \dots & \dots & \mathbf{E}_{kk} \end{pmatrix}$$

is nonsingular. If $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$, $\mathcal{P} \in \mathsf{MQ}_{n,m,q}(\mathbf{O})$ and $\{\mathbf{v}_i\}_{i\in[k]}$ in $\mathbb{F}_q^{n-m} \times \{0\}^m$ are chosen uniformly at random, then as a function of $\{\mathbf{o}_i\}_{i\in[k]} \in O$ the affine map

$$\mathcal{P}^{\star}(\mathbf{v}+\mathbf{o}) = \sum_{i=1}^{k} \mathbf{E}_{ii}\mathcal{P}(\mathbf{v}_i+\mathbf{o}_i) + \sum_{1\le i<j\le k} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{v}_i+\mathbf{o}_i,\mathbf{v}_j+\mathbf{o}_k)$$

has full rank except with probability bounded by $\frac{q^{k-(n-o)}}{q-1} + \frac{q^{m-ko}}{q-1}$.

Let $f(z)$ be an irreducible polynomial and let $\mathbf{Z} \in \mathbb{F}_q[z]^{k\times k}$ be a matrix as in Section 2.1.1. We instantiated the matrices $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ for $1 \le i,j \le k$ as the matrix that corresponds to multiplication by $\mathbf{Z}_{ij}$ in $\mathbb{F}_q[z]/f(z)$. The requirement that $f(z)$ does not divide the determinant of $\mathbf{Z}$ implies that the matrix $\mathbf{E}$ is non-singular, so Lemma 1 indeed applies to our instantiation of MAYO.

We prove Theorem 1 with two lemmas. The first lemma tightly reduces the EUF-CMA security of the MAYO signature scheme to the EUF-KOA security, by showing that we can simulate a signing oracle if B is sufficiently small. The second lemma concludes the proof by giving a tight reduction from the OV and MTWMQ problems to the EUF-KOA security game.

Lemma 2. *Suppose there exists an adversary $\mathcal{A}$ that runs in time T against the* EUF-CMA *security of the MAYO signature in the random oracle model with parameters as in Section 2.1.1, and which makes Q_h queries to the random oracle and Q_s queries to the signing oracle. Let* $\mathsf{B} = \frac{q^{k-(n-o)}}{q-1} + \frac{q^{m-ko}}{q-1}$ *and suppose $Q_s\mathsf{B} < 1$, then,* *there exists an adversary $\mathcal{B}$ against the* EUF-KOA *security of the MAYO signature scheme that runs in time $T + O((Q_h + Q_s)\mathit{poly}(n, m, k, q))$ with:*

$$\begin{aligned}\mathsf{Adv}^{\text{EUF-CMA}}_{n,m,o,k,q}(\mathcal{A}) \leq\ & \mathsf{Adv}^{\text{EUF-KOA}}_{n,m,o,q}(\mathcal{B})\,(1 - Q_s\mathsf{B})^{-1} + (Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}} \\ & + 3Q_h 2^{-8\mathsf{sk\_seed\_bytes}} + (Q_s + Q_h + 2)^2 2^{-8\mathsf{digest\_bytes}}\,.\end{aligned}$$

Proof. The EUF-KOA adversary $\mathcal{B}$ works as follows:

When $\mathcal{B}$ is given a public key pk, it starts simulating adversary $\mathcal{A}$ on input pk. $\mathcal{B}$ maintains a list L, which is initially empty. When $\mathcal{A}$ queries the random oracle at input M, $\mathcal{B}$ responds with $\mathbf{t}$ if there is an entry $(\mathsf{M}, \mathbf{t}, \star) \in L$; otherwise, $\mathcal{B}$ forwards the query to the SHAKE256 oracle, receives the response $\mathbf{t}$ from it, adds $(\mathsf{M}, \mathbf{t}, \perp)$ to L and responds to $\mathcal{A}$ with $\mathbf{t}$.

$\mathcal{B}$ chooses a random $\mathsf{seed}'_{\mathsf{sk}} \in \mathcal{B}^{\mathsf{sk\_seed\_bytes}}$. When $\mathcal{A}$ makes a query to sign a message M, $\mathcal{B}$ queries the SHAKE256 oracle on input M to get the digest M_digest and adds $(\mathsf{M}, \mathsf{M\_digest}, \perp)$ to L. Then, $\mathcal{B}$ chooses a randomizer R like in the real signing algorithm (either at random or as $\mathsf{R} = 0_{\mathsf{R\_bytes}}$) and sets $\mathsf{salt} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{R} \,\|\, \mathsf{seed}'_{\mathsf{sk}}, \mathsf{salt\_bytes})$. $\mathcal{B}$ aborts if there is an existing entry $(\mathsf{M\_digest}\|\mathsf{salt}, \mathbf{t}, \perp)$ in L. If there is an entry $(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \mathbf{t}, \mathbf{s})$ in L, then $\mathcal{B}$ answers with the signature $\mathsf{Encode}_{vec}(\mathbf{s}) \,\|\, \mathsf{salt}$. Otherwise, $\mathcal{B}$ samples $\mathbf{s} \in \mathbb{F}_q^{kn}$, and sets $\mathbf{t} = \mathcal{P}^\star(\mathbf{s})$. Then, $\mathcal{B}$ adds $(\mathsf{M\_digest}\|\mathsf{salt}, \mathbf{t}, \mathbf{s})$ to L and outputs the signature $(\mathsf{Encode}_{vec}(\mathbf{s}) \,\|\, \mathsf{salt})$. Finally, when $\mathcal{A}$ outputs a message-forgery pair (M, σ), $\mathcal{B}$ outputs the same pair.

The EUF-KOA adversary $\mathcal{B}$ runs in time $T + O((Q_h + Q_s + 1)\mathrm{poly}(n, m, k, q))$, and, hence, we only need to show that $\mathcal{B}$ succeeds in the EUF-KOA game with a sufficiently large probability. We prove this with a sequence of games starting with the EUF-CMA game played by $\mathcal{A}$ and ending with the EUF-KOA game played by $\mathcal{B}^{\mathcal{A}}$.

1. Let Game_0 be $\mathcal{A}$'s EUF-CMA game against the MAYO signature scheme. By definition, we have that $\Pr[\mathsf{Game}_0() = 1] = \mathsf{Adv}^{\text{EUF-CMA}}_{n,m,o,k,q}(\mathcal{A})$.
2. Let Game_1 be the same as Game_0 except that the game picks a second $\mathsf{seed}'_{\mathsf{sk}} \in \mathcal{B}^{\mathsf{sk\_seed\_bytes}}$, which is used instead of $\mathsf{seed}_{\mathsf{sk}}$ to derive salt when answering signing queries. If $\mathcal{A}$ does not make any random oracle queries of the form $\mathsf{M}\,\|\,\mathsf{R}\,\|\,\mathsf{seed}_{\mathsf{sk}}$ or $\mathsf{M}\,\|\,\mathsf{R}\,\|\,\mathsf{seed}'_{\mathsf{sk}}$, which happens with probability at most $2Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$ ($\mathsf{seed}_{\mathsf{sk}}$ has $8\mathsf{sk\_seed\_bytes}$ bits on min-entropy), then its view in Game_0 and Game_1 is identical. Therefore, we have $\Pr[\mathsf{Game}_1() = 1] \geq \Pr[\mathsf{Game}_0() = 1] - 2Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$.
3. Game_2 is the same as Game_1, but it simulates the random oracle SHAKE256 differently. Game_2 simulates the random oracle by maintaining a list L. When $\mathcal{A}$ makes a query on a message M, if there is an entry $(\mathsf{M}, \mathbf{t}, \star)$ in L, the game answers with $\mathbf{t}$; otherwise, it forwards the query to the SHAKE256 oracle of the EUF-CMA game to receive $\mathbf{t}$, inserts $(\mathsf{M}, \mathbf{t}, \perp)$ in L and answers with $\mathbf{t}$. When a signing query is made, Game_2 derives M_digest and salt. It then:
   - Aborts if there is an entry $(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \mathbf{t}, \perp)$ in L.
   - If there is an entry $(\mathsf{M\_digest}\|\mathsf{salt}, \mathbf{t}, \mathbf{s})$, it answers the query with the signature $\mathsf{Encode}_{vec}(\mathbf{s})\,\|$ salt (a signature derived from the found entry).
   - If there is no such entry in L, the game picks $\mathbf{t}$ uniformly at random, runs the signing algorithm for $\mathbf{t}$ to get a new $\mathbf{s}$, inserts $(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \mathbf{t}, \mathbf{s})$ in L, and outputs $\mathsf{Encode}_{vec}(\mathbf{s}) \,\|\, \mathsf{salt}$.

   Since there are at most $Q_h + Q_s$ entries of the form $(\mathsf{M}, \mathbf{t}, \perp)$ in L and there are at most Q_s signing queries, the probability of an abort is at most $Q_h Q_s 2^{-8\mathsf{salt\_bytes}}$. If the game does not abort, then it simulates the random oracle perfectly, and we have: $\Pr[\mathsf{Game}_2() = 1] \geq \Pr[\mathsf{Game}_1() = 1] - (Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}}$.
4. Game_3 is the same as Game_2, except that it answers signing queries differently. Each time a fresh signing query is made the game repeatedly picks uniformly random $\mathbf{v}_i \in \mathbb{F}_q^{n-o}$ for $i \in [k]$, until it finds a set of $\mathbf{v}_i$ such that $\mathcal{P}^*(\mathbf{v} + \cdot) : O^k \to \mathbb{F}_q^m$ has full rank. Then, instead of picking $\mathbf{t}$ at

random and sampling a random solution $\mathbf{o}$ to $\mathcal{P}^*(\mathbf{v}+\mathbf{o}) = \mathbf{t}$, Game_3 picks $\mathbf{o} \in O^k$ uniformly at random and sets $\mathbf{t} \leftarrow \mathcal{P}^*(\mathbf{v}+\mathbf{o})$. Since $\mathcal{P}^*(\mathbf{v}+\cdot)$ is a full-rank affine map, it does not affect the distribution of the signatures. The only difference is that, in Game_2, $\mathbf{v}$ and $\mathbf{o}$ are determined by the output of the SHAKE256 random oracle on input M ∥ salt ∥ $\mathsf{seed}_{\mathsf{sk}}$ ∥ ctr, whereas in Game_3 they are chosen at random. If the adversary does not make a random oracle query of the form M ∥ salt ∥ $\mathsf{seed}_{\mathsf{sk}}$ ∥ ctr (which it can do with at most a probability $Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$, since $\mathsf{seed}_{\mathsf{sk}}$ has $8\mathsf{sk\_seed\_bytes}$ bits of min-entropy), then its view in both games is the same, and we have $\Pr[\mathsf{Game}_3() = 1] \geq \Pr[\mathsf{Game}_2() = 1] - Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$.

5. Game_4 is the same as Game_3, but with a different winning condition. Game_4 outputs 0 if there are two queries to the SHAKE256 oracle that result in the same output. During the game, there are at most $Q_s + Q_h + 2$ queries (the $+2$ comes from the random oracle queries during the signature verification process) to the SHAKE256 oracle, and for each of the fewer than $(Q_s + Q_h + 2)^2$ pairs of distinct queries, the probability of a collision is $2^{-8\mathsf{digest\_bytes}}$. We, therefore, have $\Pr[\mathsf{Game}_4() = 1] \geq \Pr[\mathsf{Game}_3() = 1] - (Q_s + Q_h + 2)^2 2^{-8\mathsf{digest\_bytes}}$.

6. Game_5 is the same as Game_4, but the game picks $\mathbf{s} = \mathbf{v} + \mathbf{o}$ uniformly at random instead of $\mathbf{o}$ being random and $\mathbf{v}$ being picked uniformly at random from the set of $\mathbf{v}$ such that $\mathcal{P}^*(\mathbf{v}+\cdot)$ has full rank. Let Good_v be the event that arises when, for all the $\mathbf{v}$ chosen during the execution of the EUF-KOA game, the map $\mathcal{P}^*(\mathbf{v}+\cdot)$ happens to have full rank. Then, Game_5, conditioned on Good_v happening, is identical to Game_4. Lemma 1 states that the probability that $\mathcal{P}^*(\mathbf{v}+\cdot)$ does not have full rank for a single $\mathbf{v}$ is at most B, so, by the union bound, we have:

$$\begin{aligned}\Pr[\mathsf{Game}_5() = 1] &= \Pr[\mathsf{Game}_5() = 1 \mid \mathsf{Good\_v}] \Pr[\mathsf{Good\_v}] + \Pr[\mathsf{Game}_5() = 1 | \neg \mathsf{Good\_v}] \Pr[\neg \mathsf{Good\_v}] \\ &\geq \Pr[\mathsf{Game}_5() = 1] \Pr[\mathsf{Good\_v}] \\ &\geq \Pr[\mathsf{Game}_5() = 1](1 - Q_s \mathsf{B}) .\end{aligned}$$

7. The final game is the EUF-KOA game played by $\mathcal{B}^{\mathcal{A}}$. This is the same as Game_5, but with a different winning condition. Game_5 is won if the adversary outputs a forgery (M, sig) that is valid under the SHAKE256 oracle implemented by $\mathcal{B}$, if the signing oracle was not queried on M and if there were no collisions found in the SHAKE256 oracle. In contrast, the EUF-KOA game is only won if the forgery is valid for the SHAKE256 oracle of the EUF-KOA game. The SHAKE256 oracle implemented by $\mathcal{B}$ is the same as the oracle of the EUF-KOA game for all messages, except for the messages M_digest ∥ salt, where M_digest and salt were the message digest and salt used in one of the queries to the signing oracle. A forgery (M, sig) can only be valid for Game_4 but not for EUF-KOA game if SHAKE256(M) = SHAKE256(M′), where M′ was one of the messages queries for the signing oracle. Moreover, we must have M ≠ M′, because otherwise the forgery (M, sig) is not considered valid for Game_4, so (M, M′) is a collision for the SHAKE256 oracle. But if there was a collision, then the game would have aborted. Therefore, we have determined that if a forgery is valid for Game_5, then it must also be valid for the EUF-KOA game. So we have $\mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,q}(\mathcal{B}) \geq \Pr[\mathsf{Game}_5() = 1]$.

In case $(1 - Q_s \mathsf{B}) > 0$, we can combine the inequalities to get:

$$\begin{aligned}\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{n,m,o,k,q}(\mathcal{A}) \leq{}& \mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,q}(\mathcal{B})\,(1 - Q_s\mathsf{B})^{-1} + (Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}} \\ &+ 3Q_h 2^{-8\mathsf{sk\_seed\_bytes}} + (Q_s + Q_h + 2)^2 2^{-8\mathsf{digest\_bytes}} .\end{aligned}$$

□

Lemma 3. *Let $\mathcal{A}$ be an* EUF-KOA *adversary that runs in time T against the MAYO signature in the random oracle model with parameters as in Section 2.1.1. Then, there exists an adversary $\mathcal{B}$ against the* $\mathsf{OV}_{n,m,o,q}$ *problem, and an adversary $\mathcal{B}'$ against the* $\mathsf{MTWMQ}_{n,m,k,q}$ *problem, that both run in time bounded by $T + O((1 + Q_h)poly(n,m,k,q))$ such that:*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,k,q}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{OV}}_{n,m,o,q}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{B}') .$$

Proof. We do the proof as a short sequence of games.

1. We define Game_0 to be the EUF-KOA game played by $\mathcal{A}$. By definition, we have

$$\Pr[\mathsf{Game}_0() = 1] = \mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,k,q}(\mathcal{A}) \, .$$

2. Game_1 is the same as Game_0, except that during key generation, the challenger chooses a uniformly random $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$, instead of a $\mathcal{P}$ that vanishes on some oil space O. We construct the adversary $\mathcal{B}$ against the OV assumption as follows: when $\mathcal{B}$ is given a multivariate quadratic map $\mathcal{P}$, it computes the encodings P1_bytestring, P2_bytestring, P3_bytestring of $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$, $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$, and $\{\mathbf{P}_i^{(3)}\}_{i\in[m]}$, respectively. $\mathcal{B}$ then derives $\mathsf{seed}_{\mathsf{pk}}$ as in the normal key generation algorithm, and runs $\mathcal{A}$ on input $\mathsf{pk} = (\mathsf{seed}_{\mathsf{pk}} \| \mathsf{P3\_bytes})$, while faithfully simulating a random oracle SHAKE256, and a oracle AES-128-CTR that outputs P1_bytestring $\|$ P2_bytestring on input $\mathsf{seed}_{\mathsf{pk}}$ and that outputs random bytes otherwise. We designed $\mathcal{B}$ in such a way that, if $\mathcal{B}$ is given a $\mathcal{P} \in \mathsf{MQ}_{n,m,q}(\mathbf{O})$ for a random $\mathbf{O}$, then $\mathcal{B}^{\mathcal{A}}$ is exactly Game_0, and if $\mathcal{B}$ is given a random map $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$, then $\mathcal{B}^{\mathcal{A}}$ is Game_1. Therefore, we have:

$$\mathsf{Adv}^{\mathsf{OV}}_{n,m,o,q}(\mathcal{B}^{\mathcal{A}}) = \left|\Pr[\mathsf{Game}_0() = 1] - \Pr[\mathsf{Game}_1() = 1]\right| \, .$$

3. The next game, Game_2, is the MTWMQ game played by the adversary $\mathcal{B}'^{\mathcal{A}}$ that we now define. When $\mathcal{B}'$ is given a multivariate quadratic map $\mathcal{P}$ and oracle access to arbitrarily many random targets $\{\mathbf{t}_i\}_{i\in\mathbb{N}}$, it does the same thing as Game_1, except that instead of simulating a SHAKE256 random oracle honestly, $\mathcal{B}'$ outputs $\mathbf{t}_i$ in response to the i-th unique random oracle query, truncated or extended with random bits to achieve the requested output length. If $\mathcal{A}$ outputs a message-signature pair $(\mathsf{M}, (\mathsf{salt}, \mathbf{s}))$, then $\mathcal{B}'$ checks if the signature is valid (simulating a random oracle query in the process). If the signature is valid, then $\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt}$ is one of the random oracle queries, say the I-th unique random oracle query. Then, $\mathcal{B}'$ outputs $(I, \mathbf{s})$. If the signature is invalid, $\mathcal{B}'$ aborts. The view of $\mathcal{A}$ in this game is the same as the view of $\mathcal{A}$ in Game_1, since $\mathcal{B}'$ simulates the random oracle perfectly. Therefore, $\mathcal{A}$ outputs a valid message-signature pair with probability $\Pr[\mathsf{Game}_1() = 1]$. Therefore, we have $\mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{B}'^{\mathcal{A}}) = \Pr[\mathsf{Game}_1() = 1]$.

We can now finish the proof by combining $\Pr[\mathsf{Game}_0() = 1] = \mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,k,q}(\mathcal{A})$ with inequalities from the two game transitions to get:

$$\mathsf{Adv}^{\mathsf{EUF\text{-}KOA}}_{n,m,o,k,q}(\mathcal{A}) \le \mathsf{Adv}^{\mathsf{OV}}_{n,m,o,q}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{B}') \, .$$

□

5.3 Discussion of the advantage loss in the security proof

The security reduction from the previous section loses advantage by three additive terms $(Q_s + Q_h + 2)^2 2^{-8\mathsf{digest\_bytes}}$, $(Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}}$, and $3Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$, and one multiplicative factor $(1 - Q_s\mathsf{B})$.

Additive loss. The first two terms correspond to attacks that look for hash collisions, and that try to guess $\mathsf{seed}_{\mathsf{sk}}$ respectively. We will discuss these in Section 5.4. The remaining term $(Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}}$ corresponds to the event, in the random oracle, where the signer outputs a signature for a message (M, salt), such that the adversary has already queried the random oracle on input $\mathsf{SHAKE256}(M) \| \mathsf{salt}$. To the best of our knowledge, this term is an artifact of the proof and does not lead to an attack. Even if the salt is completely removed, there seems to be no attack. Nevertheless, to rule out any attack, we pick salt_bytes to be 24, 32, and 40 for security levels 1, 3, and 5 respectively, in order to make the term sufficiently small. Besides enabling the security proof, the salt brings some protection against fault injection and side-channel attacks.

Multiplicative loss. The security proof has a loss in advantage by a factor $(1 - Q_s\mathsf{B})$, where Q_s is the number of signatures that the EUF-CMA adversary can request, and where $\mathsf{B} = \frac{q^{k-(n-o)}}{q-1} + \frac{q^{m-ko}}{q-1}$ is an upper bound for the probability that the signer needs to restart the loop on Line 28 of MAYO.Sign. This factor stems from the fact that the rejection sampling functionality introduces a small amount of information-theoretic leakage.

If $Q_s\mathsf{B} < 1/2$, then the term only results in a loss in advantage by a constant factor, so the leakage provably does not hurt the security of MAYO by much. If $Q_s\mathsf{B} > 1$, the security proof no longer makes any guarantees, but there does not seem to be any attack that can take advantage of the information-theoretic leakage. The *Oil and Vinegar* signature scheme suffers from the same problem, but with a bigger leakage due to the larger restarting probability of approximately $1/q$. After decades of cryptanalysis, no attacks are known that can efficiently make use of this leakage. For MAYO_1, MAYO_3, and MAYO_5, the bound B is approximately 2^{-12}. For MAYO_2 the bound is approximately 2^{-20}, which means that as long as the adversary sees fewer than 2^{11} or 2^{19} signatures respectively, the leakage provably does not degrade security much. However, as explained above, we expect the MAYO signature scheme to remain secure even if the adversary has access to an unbounded number of signatures, even though the security proof no longer gives a meaningful guarantee.

5.4 Analysis of known attacks

We list the known attacks against the MAYO signature scheme, and we give estimates of their complexity. Given our security proof, we can sort attacks into three categories: attacks that exploit the losses of the security proof, attacks on the *Oil and Vinegar* problem, and attacks on the multi-target whipped MQ problem.

Table 5.1 contains lower bounds for the bit cost of the known attacks against the four proposed parameter sets. For the sake of concreteness, we say that the cost of 1 multiplication + 1 addition in $\mathbb{F}_{16}$ is 36 bit operations. This choice is arbitrary, but we make it to be consistent with the multivariate literature, where the bit-cost of one multiplication + addition in small binary fields of order 2^r is often chosen to be $2r^2 + r$ when reporting the estimated cost of attacks (see e. g., [DCP+20]). We chose the parameters such that the estimated bit costs of all the attacks exceed 2^{143}, 2^{207}, and 2^{272} for the parameter sets aiming for security levels 1, 3, and 5 respectively.

The cost of system-solving algorithms. Some of the attacks use a subroutine that finds a solution to a system of multivariate quadratic equations. We denote the bit cost of solving a random non-homogeneous system of m multivariate equations in n variables over $\mathbb{F}_q$ using the fixing Wiedemann XL algorithm [YCBC07, BFP09] by $\mathsf{XL\_Cost}_{n,m,q}$. For (over)determined systems, i.e. $n \leq m$, we can estimate this cost as:

$$\mathsf{XL\_Cost}_{n,m,q} = \min_k 36 \cdot 3 \cdot q^k \cdot \binom{n-k+D_{n-k,m}}{D_{n-k,m}}^2 \cdot \binom{n-k+2}{2},$$

where k is the number of coefficients of the solution that is guessed and that is chosen to minimize the cost, and where $D_{n-k,m}$ is the *operating degree* of XL, which can be computed as the smallest integer d for which the coefficient of t^d in the expansion of

$$\frac{(1-t^2)^m}{(1-t)^{n-k+1}}$$

is non-positive.

Note that our methodology for estimating the cost of the XL algorithm only accounts for the cost of the multiplications and ignores any other overhead such as the cost of memory access. It should therefore be interpreted as a lower bound for the cost of a realistic attack.

Table 5.1: Bit-complexity lower bounds for the state-of-the-art attacks against our proposed parameter sets. The Kipnis-Shamir, Reconciliation, and Intersection attacks are key-recovery attacks, and the Claw-finding and Direct attacks are universal forgery attacks. For the reconciliation and intersection attacks, which reduce to the hybrid XL algorithm, we report the operating degree D and the optimal number of guesses k. For the direct attack we report the ℓ and a parameter from Hashimoto's algorithm for solving underdetermined systems.

Parameter set (n, m, o, k, q)	Kipnis-Shamir	Reconciliation (D, k)	Intersection (D, k)	Direct attack (ℓ, a)	Claw-finding
MAYO$_1$ $(86, 78, 8, 10, 16)$	303	197 (17, 17)	355 (9, 1)	156 (20, 22)	169
MAYO$_2$ $(81, 64, 17, 4, 16)$	211	167 (14, 15)	228 (10, 0)	155 (14, 6)	141
MAYO$_3$ $(118, 108, 10, 11, 16)$	416	262 (23, 22)	482 (11,1)	224 (32,23)	229
MAYO$_5$ $(154, 142, 12, 12, 16)$	546	334 (30, 27)	629 (14, 0)	296 (32, 19)	298

Attacks exploiting the loss in the security proof.

Finding hash collisions. One can trivially break the EUF-CMA security of MAYO, by finding a collision for SHAKE256. If the adversary knows two messages $M_1 \neq M_2$ with SHAKE256(M_1) = SHAKE256(M_2), then it can query the signing algorithm for a signature for M_1 and output it as a forgery for M_2. Our instantiations targeting security level 1, 3, and 5 use 256, 384 and 512 bits of SHAKE256 output respectively. With these output lengths, the SHAKE256 functionality is widely believed to achieve the required security levels.

Guessing seed$_{\mathsf{sk}}$. The attacker can simply try to guess the secret key, which is a uniformly random string of sk_seed_bytes bytes. Making a correct guess would take on average approximately $2^{8\mathsf{seed}_{\mathsf{sk}}-1}$ attempts. We set sk_seed_bytes $= 24, 32$, or 40 for the parameter sets targeting security levels 1, 3, and 5 respectively. The bit length of seed$_{\mathsf{sk}}$ is longer than strictly necessary (by 64 bits) to protect against attacks that attempt to guess the secret key for one out of a large set of public keys of interest.

Attacks on the *Oil and Vinegar* problem

A MAYO public key consists of an *Oil and Vinegar* map, i.e., a multivariate quadratic map $\mathcal{P} : \mathbb{F}_{16}^n \to \mathbb{F}_{16}^m$ that vanishes on some linear subspace $O \subset \mathbb{F}_{16}^n$ of dimension o. The secret key corresponds to the space O. Therefore, an attacker can break MAYO if he can recover O from $\mathcal{P}$. This problem has been studied in the literature as it is exactly how a key recovery attack on the *Oil and Vinegar* signature scheme works (a MAYO public key is nothing but an *Oil and Vinegar* key with different parameters). We list the known attacks against this problem.

Kipnis-Shamir attack. The first attack on the *Oil and Vinegar* problem was introduced in 1998 by Kipnis and Shamir [KS98]. The attack attempts to find vectors in the oil space O, by exploiting the fact that these vectors are more likely to be eigenvectors of some publicly-known matrices. The bottleneck of the attack is computing the eigenvectors of on average q^{n-2o} matrices of size n-by-n. Asymptotically, the cost of computing the eigenvectors is the same as that of matrix multiplication. To construct Table 5.1, we use $36q^{n-2o}n^{2.8}$ as a lower bound for the bit cost of the attack. Precise estimates are not relevant because, as observed in Table 5.1, the cost of the Kipnis-Shamir attack exceeds the requirements for the claimed security level by large margins.

Reconciliation attack. [DYC⁺08] A more obvious method to find vectors in the oil space O is to use the fact that $\mathcal{P}(\mathbf{o}) = 0$ for all $\mathbf{o} \in O$. We expect random systems $\mathcal{P}$ to have approximately q^{n-m} zeros, and the *Oil and Vinegar* maps have an additional q^o artificial zeros in the subspace O. If $o = n - m$ (which is the case for the MAYO parameters), then we expect a constant fraction of the zeros of $\mathcal{P}$ to be in O, so to find a vector in O, an attacker can look for $\mathbf{x}$ such that $\mathcal{P}(\mathbf{x}) = 0$ using generic system solving algorithms. The attacker can use o random affine constraints to eliminate o variables in the system $\mathcal{P}(\mathbf{x}) = 0$, and with constant probability, the resulting system will have a unique solution, which corresponds to a vector in O. Therefore, finding a vector in O reduces to solving a system of m inhomogeneous multivariate quadratic equations in $n-o$ variables. Once a single vector in O is found, finding the rest of O is a much easier problem, so the cost of the reconciliation attack is $\mathsf{XL\_Cost}_{m,n-o,q}$.

Intersection attack The intersection attack, introduced by Beullens [Beu21] is a generalization of the reconciliation attack which uses the ideas behind the Kipnis-Shamir attack. The idea is to simultaneously look for more than one vector in the oil space. Let $k \geq 2$ be some parameter, then the attack tries to find k vectors in O by solving a system of $M = \binom{k+1}{2}m - 2\binom{k}{2}$ quadratic equations in $N = \min(n, nk - (2k-1)m)$ variables. In the context of MAYO, we get the most efficient attacks in the case $k = 2$. The attack is only guaranteed to work if $3o > n$, which is not the case for the MAYO parameters. If $3o \leq n$, then the attack succeeds with probability $q^{-n+3o-1}$, so the attack needs to be repeated on average q^{n-3o+1} times, which makes the cost of the attack:

$$q^{n-3o+1}\mathsf{XL\_Cost}_{3m-2,n,q}\,.$$

Because o is very small in MAYO, the intersection attack has a very low success probability. This makes the attack much less efficient compared to the common Oil-and-Vinegar setting where $o = m$.

Rectangular Minrank attack Furue and Ikematsu showed that the rectangular minrank attack is applicable to MAYO [FI23]. The idea behind rectangular Minrank attacks [Beu21] is to look at linear maps of the form $L_{\mathbf{x}} : \mathbb{F}_{16}^n \to \mathbb{F}_{16}^m : \mathbf{y} \mapsto P'(\mathbf{x}, \mathbf{y})$. It follows from the fact that $\mathcal{P}$ vanishes on O, that for any $\mathbf{o} \in O$, the map $L_{\mathbf{o}}$ has rank at most $n - o$, because O is included in its kernel. Therefore, if $n - o < m$, then $L_{\mathbf{o}}$ has an unusually small rank, since a random linear map from $\mathbb{F}_{16}^n$ to $\mathbb{F}_{16}^m$ has rank close to m. A strategy to find vectors in O is therefore to look for vectors $\mathbf{x}$ such that $L_{\mathbf{x}}$ has low rank, which is an instance of a the minrank problem. For the round 1 parameters of MAYO, this attack is slightly more costly than other key recovery attacks, so the attack does not directly threaten the security of the MAYO. For more details we refer to [FI23]. In the round 2 version of MAYO we propose parameters such that $n - o \geq m$, which completely prevents the attack, since $L_{\mathbf{o}}$ will have full rank for most $\mathbf{o} \in O$.

Attacks on the multi-target whipped MQ problem.

A signature for a message M consists of a vector $\mathbf{s} \in \mathbb{F}_{16}^{n \times k}$ and a salt salt $\in \mathcal{B}^{\mathsf{salt\_bytes}}$ such that $\mathcal{P}^*(\mathbf{s}) =$ SHAKE256(SHAKE256(M) ∥ salt), where

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

is the whipped *Oil and Vinegar* map. Therefore, an attacker can forge a signature for a message M by hashing M with many salts and then trying to find a vector $\mathbf{s}$ such that $\mathcal{P}^*(\mathbf{s})$ equals one of the hashes. Here we give the best known ways an attacker could do this.

Direct attack. In a direct attack the attacker picks a random salt salt $\in \mathcal{B}^{\text{salt\_bytes}}$, and solves for $\mathbf{s} \in \mathbb{F}_{16}^{n \times k}$ such that $\mathcal{P}^*(\mathbf{s}) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M) \parallel \mathsf{salt})$. There are at present no known algorithms that can take advantage of the structure of the system $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}$ to find a solution more efficiently. Therefore, to estimate the cost of this attack we will assume that $\mathcal{P}^*(\mathbf{s}) = \mathbf{t}$ behaves like a generic system of m quadratic equations in kn variables.

The system $\mathcal{P}(\mathbf{s}) = \mathbf{t}$ is underdetermined, i.e. the number of variables nk is larger than the number of equations m, and therefore the system is expected to have many solutions. The problem of finding a solution to an underdetermined system of equations clearly reduces to the problem of solving one of many determined systems: by fixing $nk - m$ variables in $\mathcal{P}(\mathbf{s}) = \mathbf{t}$ one gets a smaller determined system $\mathcal{P}^*(\mathbf{s}^*) = \mathbf{t}$, and any solution to such a determined system yields a solution to $\mathcal{P}(\mathbf{s}) = \mathbf{t}$. A line of work by Thomae, Wolf, Furue, Nakamura, Takagi, Hashimoto and others [TW12, FNT21, Has21] has shown that if a system of quadratic equations is sufficiently underdetermined, then one can improve over this naive appraoach by doing a change of variables on $\mathbf{s}$, and then fixing variables more cleverly.

Concretely, for parameters a and ℓ satisfying $nk > (a+1)(m-\ell-a+1)$ and $nk > a(m-\ell)-(a-1)^2+\ell$, the method of Hashimoto [Has21] finds a solution to $\mathcal{P}(\mathbf{s}) = \mathbf{t}$ with an estimated cost of

$$q^{\ell}\left(\mathsf{XL\_Cost}_{m-a-\ell,m-a,q} + \mathsf{XL\_Cost}_{a-1,a-1,q}\right) + (m-a-\ell+1)\mathsf{XL\_Cost}_{a,a,q}\ ,$$

which is the formula we used to give a lower bound for the bit-cost of a direct attack against MAYO in Table 5.1.

Claw finding attacks. An attacker can forge a signature for message M by finding a salt and $\mathbf{s}$ such that $\mathcal{P}(\mathbf{s}) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt})$, which is a claw-finding problem. Since $\mathcal{P}$ is homogeneous it suffices to find a collision up to a scalar multiple, i.e. find salt and $\mathbf{s}$ for which $\alpha\mathcal{P}(\mathbf{s}) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt})$ for some $\alpha \in \mathbb{F}_{16}$, because then $(\mathsf{salt}, \sqrt{\alpha}\mathbf{s})$ is a valid forgery. [1] We give a lower bound for the cost of finding such a claw.

To optimize the attack, one can evaluate only the first m' equations of $\mathcal{P}$ (as opposed to all m of them), and look for partial collisions. Then, for each partial collision $(\mathsf{salt}, \mathbf{s})$ we fully evaluate $\mathcal{P}(\mathbf{s})$ to check if it is a full collision. The expected number of false positives is $q^{m-m'}$, so if we put $m' \gtrapprox m/2$, then the number of false positives is small enough so that the cost of checking the false positives is negligible so that we get a constant factor speedup of $m'/m \approx 1/2$.

There are $q^m/(q-1)$ nonzero vectors in $\mathbb{F}_{16}^m$ up to a scalar, so an attacker that computes $\mathcal{P}(\mathbf{s}_i)$ for X inputs $\{\mathbf{s}_i\}_{i\in[X]}$ and computes $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt}_j)$ for Y salts $\{\mathsf{salt}_j\}_{j\in[Y]}$ finds on average $XYq^{-m}(q-1)$ collisions. Using Gray code enumeration [BCC+10], one can evaluate the first $m' \leq m$ polynomials of $\mathcal{P}$ in many inputs at an amortized cost of $2\log(q)m'$ bit operations per evaluation. For the sake of concreteness, we say that computing SHAKE256 has a bit cost of at least $2^{17.2}$ [Sma]. To find a collision with probability $\approx e^{-1}$ we need $XY \approx q^m/(q-1)$. A lower bound for the bit-cost of the attack, divided by the success probability is then:

$$e\left(2\log(q)m'X + 2^{17.2}Y\right)\ ,$$

which is equal to $2^{11.0}\sqrt{m\log(q)q^m/(q-1)}$ for $m' = m/2$ and optimally chosen X, Y such that $XY = q^m/(q-1)$, which is the formula we use in Table 5.1. Note that for MAYO_2 we get a lower bound of 2^{141} bit operations, which is slightly lower than the estimate of 2^{143} gates for a key search on AES128 mentioned in the NIST call for proposals document. Nevertheless, we claim MAYO_2 reaches NIST security level 1, since we have ignored significant overheads. The analysis outlined above only looks at the cost of evaluting $\mathcal{P}$ and SHAKE256, ignoring the cost of actually extracting the collision from the evaluations, which would cause some overhead. E.g., a naive implementation based on large lists of evaluations incurs a large cost of accessing memory, while a more realistic attacker which uses a memoryless claw-finding algorithm such as Pollard's rho or van Oorschot-Wiener [vOW96] would likely not be able to take full advantage of the Gray code enumeration technique.

[1] We thank M. Saarinen for pointing this out on the NIST PQC forum.

Remark. *We note that it would be possible to increase the cost of claw finding attacks by a few bits "for free", by using a slower hash function to derive the target* $\mathfrak{t}$*. Using a hash function that is slower by a factor* X*, increases the cost of the attack by a factor* $\sqrt{X}$*. For small enough values of* X*, e.g.* $X = 256$*, the cost of the hash function remains negligible compared to the total cost of the signing and verification procedures. We did not deem this modification necessary.*

Quantum attacks.

All the known quantum attacks against MAYO are obtained by speeding some part of a classical attack up with Grover's algorithm. Therefore, they outperform the classical attacks by at most a square root factor, and they do not threaten our security claims. Indeed, the NIST security levels 1,3, and 5 are defined with respect to the hardness of a key search against a block cipher such as the AES with $128, 192$, or 256-bit keys respectively. Grover speeds up a key search by almost a square root factor, so, for a quantum attack to break the NIST security targets it needs to improve on classical attacks by more than a square root factor, which is not possible by relying on Grover's algorithm alone.

We very briefly discuss how the different attacks can be sped up by Grover's algorithm:

Claw finding and Hash collisions. Claw-finding and collision-finding for functions that are cheap to compute are not believed to benefit from quantum computing [JS19].

System-solving attacks. The attacks that reduce to system-solving such as the direct attack, the reconciliation attack, and the intersection attack benefit relatively little from Grover's algorithm, because only a small part of the cost comes from guessing some of the variables, and only this part can be sped up with Grover's algorithm.

Kipnis-Shamir attack. Almost all of the cost of the Kipnis-Shamir attack comes from guessing a certain matrix in the hope that it has a good eigenvector, so here Grover can almost fully achieve a quadratic speedup (assuming there is no restriction on the depth of a quantum attack.). However, for our proposed parameters the Kipnis-Shamir attack is much less efficient than the relevant key search against the AES classically, and the depth of the Grover oracle that checks if a matrix has a good eigenvalue is larger than the depth of a Grover oracle that checks if an AES key-guess is correct. Therefore, a Groverized Kipnis-Shamir attack against $\mathsf{MAYO}_1/\mathsf{MAYO}_2$, MAYO_3, or MAYO_5, is much more costly than a Groverized key search against AES-128, AES-192, or AES-256 respectively.

Guessing $\mathsf{seed}_{\mathsf{sk}}$. One can almost fully achieve a quadratic speedup for the $\mathsf{seed}_{\mathsf{sk}}$-guessing attack, but we choose the length of $\mathsf{seed}_{\mathsf{sk}}$ to be 64 bits longer than the length of the AES key that defines the claimed security level (e. g., $\mathsf{seed}_{\mathsf{sk}}$ has 192 bits for SL 1 which is defined with respect to AES with 128-bit keys), so this also does not threaten the security claim.

Chapter 6

Advantages and Limitations

Advantages

Small key and signature sizes. Compared to other post-quantum digital signature algorithms, the MAYO signature scheme has short keys and very short signatures.

Computational efficiency. MAYO offers good performance for key generation, signing, and verification. The performance of our optimzed implementations of MAYO is similar to that of lattice-based signatures on modern Intel x84-64 or ARM CPUs, and is only slower by a small factor on embedded platforms such as Cortex-M4 CPUs.

Flexible. Parameter sets are easily adjusted to reach a specific security level. For each target security level, there is a flexible trade-off between signature size and public key size, as demonstrated in Table 2.2.

Wide security margin against known attacks. State-of-the-art attacks against MAYO are well-understood and easy to analyze. We pick parameters using a conservative methodology that only focuses on gate count and ignores the cost of memory accesses and parallelization. Moreover, for the SL1, SL3, and SL5 parameters we ensured a margin of 10, 15, and 20 bits of security respectively against system-solving attacks, to hedge against future improvements in generic system solving techniques.

Limitations

Scalability to higher security levels. Multivariate quadratic maps need $\mathcal{O}(\lambda^3)$ coefficients to reach $O(\lambda)$ bits of security. This causes multivariate cryptosystems, such as MAYO, to scale less well to higher security levels, compared to e. g., lattice-based signature schemes. For example, even though at the lowest security level the combined public key and signature size of MAYO is only 50% of that of the Dilithium scheme, at security level 5, the combined size of MAYO is already 90% of that of Dilithium. At sufficiently higher security levels Dilithium would become more compact than MAYO.

New design. MAYO, invented in 2021, is a relatively recent design. MAYO public keys have the same structure as Oil and Vinegar public keys, so decades of cryptanalysis inspire confidence in the security of MAYO against key recovery attacks. However, for security against forgery attacks, MAYO relies on the hardness of the "Whipped MQ" problem, which has had relatively less public scrutiny.

Bibliography

[AES01] Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce, November 2001.

[BCC$^+$10] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 203–218. Springer, Berlin, Heidelberg, August 2010.

[BCC$^+$24] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized implementations for AVX2 and cortex-M4. *IACR TCHES*, 2024(2):252–275, 2024.

[Beu21] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 348–373. Springer, Cham, October 2021.

[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, Cham, September / October 2022.

[BFP09] Luk Bettale, Jean-Charles Faugere, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, 3(3):177–197, 2009.

[BR98] Mihir Bellare and Phillip Rogaway. Pss: Provably secure encoding method for digital signatures. *IEEE P1363a: Provably secure signatures*, 1998.

[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`.

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

[DYC$^+$08] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of Rainbow. In Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung, editors, *ACNS 08International Conference on Applied Cryptography and Network Security*, volume 5037 of *LNCS*, pages 242–257. Springer, Berlin, Heidelberg, June 2008.

[FI23] Hiroki Furue and Yasuhiko Ikematsu. A new security analysis against MAYO and QR-UOV using rectangular MinRank attack. In Junji Shikata and Hiroki Kuzuno, editors, *IWSEC 23*, volume 14128 of *LNCS*, pages 101–116. Springer, Cham, August 2023.

[FNT21] Hiroki Furue, Shuhei Nakamura, and Tsuyoshi Takagi. Improving Thomae-Wolf algorithm for solving underdetermined multivariate quadratic polynomial problem. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*, pages 65–78. Springer, Cham, 2021.

[Has21] Yasufumi Hashimoto. An improvement of algorithms to solve under-defined systems of multivariate quadratic equations. *Cryptology ePrint Archive*, 2021.

[JS19] Samuel Jaques and John M. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 32–61. Springer, Cham, August 2019.

[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, Berlin, 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.

[KS98] Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil & Vinegar signature scheme. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 257–266. Springer, Berlin, Heidelberg, August 1998.

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

[SHA15] Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[Sma] Nigel Smart. 'Bristol Fashion' MPC Circuits. https://nigelsmart.github.io/MPC-Circuits/.

[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, Cham, August 2016.

[TW12] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 156–171. Springer, Berlin, Heidelberg, May 2012.

[vOW96] Paul C. van Oorschot and Michael J. Wiener. Improving implementable meet-in-the-middle attacks by orders of magnitude. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 229–236. Springer, Berlin, Heidelberg, August 1996.

[YCBC07] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J Bernstein, and Jiun-Ming Chen. Analysis of quad. In *Fast Software Encryption: 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers 14*, pages 290–308. Springer, 2007.