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 $\mathcal{P}(\mathbf{x}) = (p_1, \ldots, p_m) : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ 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 quadratic maps $\mathcal{P}$ we can define it's 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

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

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 $\mathcal{P}^*(\mathbf{x}_1, \ldots, \mathbf{x}_k) : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$, 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 $O^k = \{(\mathbf{o}_1, \ldots, \mathbf{o}_k) \,|\, \forall i : \mathbf{o}_i \in O\}$ which has dimension ko. Concretely, we define:

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

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 $(\mathbf{v}_1, \ldots, \mathbf{v}_k) \in \mathbb{F}_q^{kn}$ at random, and then solves for $(\mathbf{o}_1, \ldots \mathbf{o}_k) \in O^k$ such that

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

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 $(n, m, o, k, q) =$ (66,64,8,9,16), which results in a public key of 1168 bytes, and a signature size of 321 bytes (297 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 58-fold reduction in public key size at the cost of a 3-fold increase in signature size.

Changes with respect to [Beu22].

- **New parameter choices.** We propose new parameter choices that allow for simple and optimized implementations of *MAYO*. In particular, we choose the m parameter to be a multiple of 32.
- **Bitsliced encoding of the private and public key.** We store and sample the coefficients of $\mathcal{P}(\mathbf{x})$ in a bitsliced form, which allows doing most of the $\mathbb{F}_q$-arithmetic in *MAYO* using a bitsliced approach, which is particularly efficient on low-end devices where powerful SIMD instructions are not available.

[1] For security reasons, we choose these matrices to have the property that all their non-trivial linear combinations have rank 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 a multiple of 32.
- 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 $x \overset{\$}{\leftarrow} X$ to denote that x is assigned a value chosen from X uniformly at random. If A is an algorithm, we write $x \leftarrow A(y)$ to denote that x is assigned the output of running A on input y. If k is an integer, we denote by $[k]$ the set $\{0,\ldots,k-1\}$. We denote by $\{x_i\}_{i\in[k]}$ a sequence of objects $x_0,\ldots,x_{k-1}$ indexed by elements of $[k]$. We denote the base-2 logarithm by $\log$, and we denote binomial coefficients by $\binom{n}{k}$, i.e., $\binom{n}{k} = n!/k!(n-k)!$.

Bytes and byte strings. Inputs and outputs to all *MAYO* API functions are byte strings. We denote by $\mathcal{B} = [256] = \{0,\ldots,255\}$ 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 $\mathbf{x}_i,\ldots,\mathbf{x}_{j-1}$. 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 $\mathbf{A}^\mathsf{T}$ the transpose of $\mathbf{A}$, i.e. the matrix in $\mathbb{F}_{16}^{n\times m}$ such that $\mathbf{A}^\mathsf{T}[i,j] = \mathbf{A}[j,i]$ for all $0 \le j < m$ and $0 \le i < n$.

We define the function $\mathsf{Upper} : \mathbb{F}_q^{n\times n} \to \mathbb{F}_q^{n\times n}$ 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 $\mathsf{SampleSolution}(\mathbf{A},\mathbf{y},\mathbf{r})$ (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 $\mathbf{Ax}' = y - \mathbf{A}r$ 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 $\mathsf{SampleSolution}(\mathbf{A},\mathbf{y},\mathbf{r})$ outputs $\perp$. 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 $\mathsf{EF}(\mathbf{B})$ 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 $\mathsf{EF}(\mathbf{B})$

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 $\mathbf{B}$.

```
pivot_row ← $0$, pivot_column ← $0$
while pivot_row $< m$ and pivot_column $< ko + 1$ do
    possible_pivots ← $\{i \mid h \le i < m$ and $\mathbf{B}[i,k] \ne 0\})$
    if possible_pivots $= \emptyset$ then
        pivot_column ← pivot_column $+ 1$                // Move to next column if there is no pivot.
        continue
    next_pivot_row ← $\min$(possible_pivots)
    Swap($\mathbf{B}$[pivot_row, :], $\mathbf{B}$[next_pivot_row, :])

    //Make the leading entry a "1".
    $\mathbf{B}$[pivot_row, :] ← $\mathbf{B}$[pivot_row, pivot_column]$^{-1}$$\mathbf{B}$[pivot_row, :]

    //Eliminate entries below the pivot.
    for row from next_pivot_row to $m - 1$ do
        $\mathbf{B}$[row, :] ← $\mathbf{B}$[row, :] − $\mathbf{B}$[row, pivot_column]$\mathbf{B}$[pivot_row, :]

    pivot_row ← pivot_row $+ 1$
    pivot_column ← pivot_column $+ 1$
return 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 $\perp$.
//Randomize the system using $\mathbf{r}$.
$\mathbf{x} \leftarrow \mathbf{r} \in \mathbb{F}_q^{ko}$
$\mathbf{y} \leftarrow \mathbf{y} - \mathbf{Ar}$

//Put $(\mathbf{A}\,\mathbf{y})$ in echelon form with leading 1's.
$(\mathbf{A}\,\mathbf{y}) \leftarrow \mathsf{EF}((\mathbf{A}\,\mathbf{y}))$

//Check if $\mathbf{A}$ has rank $m$.
if $\mathbf{A}[m-1, :] = \mathbf{0}_n$ then
    return $\perp$

//Back-substitution
for $r$ from $m-1$ to 0 do
    //Let $c$ be the index of first non-zero element of $A[r, :]$.
    $\mathbf{x}_c \leftarrow \mathbf{x}_c + \mathbf{y}_r$
    $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{y}_r A[:, c]$
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 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 $\mathsf{Encode}_{\mathbb{F}_{16}}(x_1), \dots, \mathsf{Encode}_{\mathbb{F}_{16}}(x_n)$, 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 $\mathbf{v} = (\mathbf{O}[0,:]\ \mathbf{O}[1,:]\ \dots\ \mathbf{O}[n-o-1,:])$ 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 Bitsliced encodings of m (upper triangular) matrices.

Algorithm 3 defines a bitsliced encoding function $\mathsf{EncodeBitslicedMatrices}$ for the sets of m matrices that can be used in implementations that use efficient bitsliced arithmetic. This function encodes a sequence of Matrices $\mathbf{A}_0, \dots, \mathbf{A}_{m-1} \in \mathbb{F}_q^{r\times c}$.

It starts by encoding the m field elements $\mathbf{A}_0[0,0], \mathbf{A}_1[0,0], \dots, \mathbf{A}_{m-1}[0,0]$ in a bitsliced format, using $\mathsf{EncodeBitslicedVector}$ (as defined in Algorithm 4). Then, it encodes the $\mathbf{A}_i[0,1]$ entries up to the $\mathbf{A}_i[0,c-1]$ entries, after which we encode the $\mathbf{A}_i[1,0]$ entries until, we finish with the encoding of the $\mathbf{A}_i[r-1,c-1]$ entries.

If the $\mathbf{A}_i$ matrices are upper triangular, then $\mathsf{EncodeBitslicedMatrices}$ 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 $\mathsf{EncodeBitslicedMatrices}(r, c, \{\mathbf{A}\}_{i\in[m]}, \mathsf{is\_triangular})$

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.

```
bytestring = $\emptyset$
for $i$ from 0 to $r-1$ do
    for $j$ from 0 to $c-1$ do
        if $i \le j$ or is_triangular = false then
            bytestring = bytestring||EncodeBitslicedVector($\{\mathbf{A}_k[i,j]\}_{k\in[m]}$)
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. $\mathsf{Encode}_{\mathbf{P}^{(1)}}(\cdot) := \mathsf{EncodeBitslicedMatrices}(n-o, n-o, \cdot, \mathsf{true})$
2. $\mathsf{Encode}_{\mathbf{P}^{(2)}}\cdot) := \mathsf{EncodeBitslicedMatrices}(n-o, o, \cdot, \mathsf{false})$
3. $\mathsf{Encode}_{\mathbf{P}^{(3)}}(\cdot) := \mathsf{EncodeBitslicedMatrices}(o, o, \cdot, \mathsf{true})$

Algorithm 4 EncodeBitslicedVector($\mathbf{v}$)

Input: A vector $\mathbf{v} \in \mathbb{F}_{16}^m$

Output: A byte string bytestring $\in \mathcal{B}^{m/2}$ that encodes $\mathbf{v}$ in a bitsliced format.

```
bytestring ← $0, \cdots, 0 \in \mathcal{B}^{m/2}$
**for** $i$ from 0 to $(m/8) - 1$ **do**
    //Encode 8 elements of $\mathbf{v}$ into 4 bytes.
    $b_0 \leftarrow 0, b_1 \leftarrow 0, b_2 \leftarrow 0, b_3 \leftarrow 0$
    **for** $j$ from 7 to 0 **do**
        Let $\mathbf{v}[i * 8 + j] = a_0 + a_1 x + a_2 x^2 + a_3 x^3$
        $b_0 \leftarrow b_0 * 2 + a_0$
        $b_1 \leftarrow b_1 * 2 + a_1$
        $b_2 \leftarrow b_2 * 2 + a_2$
        $b_3 \leftarrow b_3 * 2 + a_3$

    //Write the 4 bytes to their position in the byte-string.
    bytestring$[i] \leftarrow b_0$
    bytestring$[m/8 + i] \leftarrow b_1$
    bytestring$[2 * m/8 + i] \leftarrow b_2$
    bytestring$[3 * m/8 + i] \leftarrow b_3$
**return** bytestring.
```

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 5): 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 6): 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 7): 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 8): 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 9): 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.

Remark. *In lines 30,31, and 33 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 $\mathbb{F}_q[z]/f(z)$. 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 29 of MAYO.Sign.*

Algorithm 5 MAYO.CompactKeyGen()

```
Output: Compact representation of a secret key $\mathsf{csk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$
Output: Compact representation of a public key $\mathsf{cpk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$
//Pick $\mathsf{seed}_{\mathsf{sk}}$ at random.
$\mathsf{seed}_{\mathsf{sk}} \xleftarrow{\$} \mathcal{B}^{\mathsf{sk\_seed\_bytes}}$

//Derive $\mathsf{seed}_{\mathsf{pk}}$ and $\mathbf{O}$ from $\mathsf{seed}_{\mathsf{sk}}$.
$\mathsf{S} \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_{\mathsf{sk}}, \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes})$
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \mathsf{S}[0 : \mathsf{pk\_seed\_bytes}]$    // $\mathsf{seed}_{\mathsf{pk}} \in \mathcal{B}^{\mathsf{pk\_seed\_bytes}}$
$\mathbf{O} \leftarrow \mathsf{Decode}_{\mathbf{O}}(\mathsf{S}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes}])$    // $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$

//Derive the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$.
$\mathsf{P} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_{\mathsf{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes})$
$\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P}[0 : \mathsf{P1\_bytes}])$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
$\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P}[\mathsf{P1\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes}])$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$

//Compute the $\mathbf{P}_i^{(3)}$.
for $i$ from 0 to $m-1$ do
    $\mathbf{P}_i^{(3)} \leftarrow \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$    // $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ upper triangular

//Encode the $\mathbf{P}_i^{(3)}$.
$\mathsf{cpk} \leftarrow \mathsf{seed}_{\mathsf{pk}} \parallel \mathsf{Encode}_{\mathbf{P}^{(3)}}(\{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
$\mathsf{csk} \leftarrow \mathsf{seed}_{\mathsf{sk}}$


//Output keys.
return $(\mathsf{cpk}, \mathsf{csk})$.
```

Algorithm 6 MAYO.ExpandSK(csk)

```
Input: Compacted secret key $\mathsf{csk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$
Output: Expanded secret key $\mathsf{esk} \in \mathcal{B}^{\mathsf{esk\_bytes}}$
//Parse csk
$\mathsf{seed}_{\mathsf{sk}} \leftarrow \mathsf{csk}[0 : \mathsf{sk\_seed\_bytes}]$

//Derive $\mathsf{seed}_{\mathsf{pk}}$ and $\mathbf{O}$ from $\mathsf{seed}_{\mathsf{sk}}$.
$\mathsf{S} \leftarrow \mathsf{SHAKE256}(\mathsf{seed}_{\mathsf{sk}}, \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes})$
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \mathsf{S}[0 : \mathsf{pk\_seed\_bytes}]$    // $\mathsf{seed}_{\mathsf{pk}} \in \mathcal{B}^{\mathsf{pk\_seed\_bytes}}$
$\mathsf{O\_bytestring} \leftarrow \mathsf{S}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{O\_bytes}]$    // $\mathsf{O\_bytestring} \in \mathcal{B}^{\mathsf{O\_bytes}}$
$\mathbf{O} \leftarrow \mathsf{Decode}_{\mathbf{O}}(\mathsf{O\_bytestring})$    // $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$

//Derive the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$.
$\mathsf{P} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_{\mathsf{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes})$
$\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P}[0 : \mathsf{P1\_bytes}])$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
$\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P}[\mathsf{P1\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes}])$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$

//Compute the $\mathbf{L}_i$.
for $i$ from 0 to $m-1$ do
    $\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)}$    // $\mathbf{L}_i \in \mathbb{F}_q^{(n-o)\times o}$

//Encode the $\mathbf{L}_i$ and output esk.
return $\mathsf{esk} = \mathsf{seed}_{\mathsf{sk}} \parallel \mathsf{O\_bytestring} \parallel \mathsf{P}[0 : \mathsf{P1\_bytes}] \parallel \mathsf{Encode}_{\mathbf{L}}(\{\mathbf{L}_i\}_{i\in[m]})$.
```

Algorithm 7 MAYO.ExpandPK(cpk)

```
Input: Compact public key $\mathsf{cpk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$
Output: Expanded public key $\mathsf{epk} \in \mathcal{B}^{\mathsf{epk\_bytes}}$
//Parse cpk.
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \mathsf{cpk}[0 : \mathsf{pk\_seed\_bytes}]$

//Expand $\mathsf{seed}_{\mathsf{pk}}$ and return.
$\mathsf{epk} = \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}_{\mathsf{pk}}, \mathsf{P1\_bytes} + \mathsf{P2\_bytes}) \,\|\, \mathsf{cpk}[\mathsf{pk\_seed\_bytes} : \mathsf{pk\_seed\_bytes} + \mathsf{P3\_bytes}]$
return epk.
```

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 5): 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 10): 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 11): 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 8 MAYO.Sign(esk, M)

```
Input: Expanded secret key esk $\in \mathcal{B}^{\mathsf{esk\_bytes}}$
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 $\in \mathcal{B}^{\mathsf{sig\_bytes}}$
//Decode esk.
$\mathsf{seed_{sk}} \leftarrow \mathsf{esk}[0 : \mathsf{sk\_seed\_bytes}]$
$\mathbf{O} \leftarrow \mathsf{Decode}_{\mathbf{O}}(\mathsf{esk}[\mathsf{sk\_seed\_bytes} : \mathsf{sk\_seed\_bytes} + \mathsf{O\_bytes}])$
$\{\mathbf{P}_i^{(1)}\}_{i \in m} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{esk}[\mathsf{sk\_seed\_bytes} + \mathsf{O\_bytes} : \mathsf{sk\_seed\_bytes} + \mathsf{O\_bytes} + \mathsf{P1\_bytes}])$
$\{\mathbf{L}_i\}_{i \in m} \leftarrow \mathsf{Decode}_{\mathbf{L}}(\mathsf{esk}[\mathsf{sk\_seed\_bytes} + \mathsf{O\_bytes} + \mathsf{P1\_bytes} : \mathsf{esk\_bytes}])$    // $\mathbf{L}_i \in \mathbb{F}_q^{(n-o) \times o}$

//Hash message and derive salt and $\mathbf{t}$.
$\mathsf{M\_digest} \leftarrow \mathsf{SHAKE256}(M, \mathsf{digest\_bytes})$    // $\mathsf{M\_digest} \in \mathcal{B}^{\mathsf{digest\_bytes}}$
$\mathsf{R} \leftarrow 0_{\mathsf{R\_bytes}}$ or $\mathsf{R} \xleftarrow{\$} \mathcal{B}^{\mathsf{R\_bytes}}$    // Optional randomization
$\mathsf{salt} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{R} \,\|\, \mathsf{seed_{sk}}, \mathsf{salt\_bytes})$    // $\mathsf{salt} \in \mathcal{B}^{\mathsf{salt\_bytes}}$
$\mathbf{t} \leftarrow \mathsf{Decode}_{vec}(m, \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \lceil m \log(q)/8 \rceil))$    // $\mathbf{t} \in \mathbb{F}_q^m$

//Attempt to find a preimage for $\mathbf{t}$.
for ctr from 0 to 255 do
    //Derive $\mathbf{v}_i$ and $\mathbf{r}$.
    $\mathsf{V} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{salt} \,\|\, \mathsf{seed_{sk}} \,\|\, \mathsf{ctr}, k * \mathsf{v\_bytes} + \lceil ko \log(q)/8 \rceil)$
    for $i$ from 0 to $k-1$ do
        $\mathbf{v}_i \leftarrow \mathsf{Decode}_{vec}(n-o, \mathsf{V}[i * \mathsf{v\_bytes} : (i+1) * \mathsf{v\_bytes}])$    // $\mathbf{v}_i \in \mathbb{F}_q^{n-o}$
    $\mathbf{r} \leftarrow \mathsf{Decode}_{vec}(ko, \mathsf{V}[k * \mathsf{v\_bytes} : k * \mathsf{v\_bytes} + \lceil ko \log(q)/8 \rceil])$

    //Build linear system $\mathbf{Ax} = \mathbf{y}$.
    $\mathbf{A} \leftarrow 0_{m \times ko} \in \mathbb{F}_q^{m \times ko}$
    $\mathbf{y} \leftarrow \mathbf{t}, \ell \leftarrow 0$
    for $i$ from 0 to $k-1$ do
        $\mathbf{M}_i \leftarrow \mathbf{0}_{m \times o} \in \mathbb{F}_q^{m \times o}$
        for $j$ from 0 to $m-1$ do
            $\mathbf{M}_i[j, :] \leftarrow \mathbf{v}_i^\mathsf{T} \mathbf{L}_j$    // Set $j$-th row of $\mathbf{M}_i$
        for $j$ from $k-1$ to $i$ do
            $\mathbf{u} = \begin{cases} \{\mathbf{v}_i^\mathsf{T} \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a \in [m]} & \text{if } i = j \\ \{\mathbf{v}_i^\mathsf{T} \mathbf{P}_a^{(1)} \mathbf{v}_j + \mathbf{v}_j^\mathsf{T} \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a \in [m]} & \text{if } i \neq j \end{cases}$    // $\mathbf{u} \in \mathbb{F}_q^m$
            $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}^\ell \mathbf{u}$
            $\mathbf{A}[:, i * o : (i+1) * o] \leftarrow \mathbf{A}[:, i * o : (i+1) * o] + \mathbf{E}^\ell \mathbf{M}_j$
            if $i \neq j$ then
                $\mathbf{A}[:, j * o : (j+1) * o] \leftarrow \mathbf{A}[:, j * o : (j+1) * o] + \mathbf{E}^\ell \mathbf{M}_i$
            $\ell \leftarrow \ell + 1$

    //Try to solve the system.
    $\mathbf{x} \leftarrow \mathsf{SampleSolution}(\mathbf{A}, \mathbf{y}, \mathbf{r})$    // $\mathbf{x} \in \mathbb{F}_q^{ko} \cup \{\perp\}$
    if $\mathbf{x} \neq \perp$ then
        break

//Finish and output the signature.
$\mathbf{s} \leftarrow \mathbf{0}_{kn}$    // $\mathbf{s} \in \mathbb{F}_q^{kn}$
for $i$ from 0 to $k-1$ do
    $\mathbf{s}[i * n : (i+1) * n] \leftarrow (\mathbf{v}_i + \mathbf{Ox}[i * o : (i+1) * o]) \,\|\, \mathbf{x}[i * o : (i+1) * o]$
return $\mathsf{sig} = \mathsf{Encode}_{vec}(\mathbf{s}) \,\|\, \mathsf{salt}$.
```

Algorithm 9 $\mathsf{MAYO.Verify}(\mathsf{epk}, \mathsf{M}, \mathsf{sig})$

```
Input: Expanded public key $\mathsf{epk} \in \mathcal{B}^{\mathsf{epk\_bytes}}$
Input: Message $\mathsf{M} \in \mathcal{B}^*$
Input: Signature $\mathsf{sig} \in \mathcal{B}^{\mathsf{sig\_bytes}}$
Constant: $\mathbf{E} \in \mathbb{F}_q^{m \times m}$ # Represents multiplication by $z$ in $\mathbb{F}_q[z]/(f(z))$
Output: An integer $\mathsf{result}$ to indicate if $\mathsf{sig}$ is valid ($\mathsf{result} = 0$) or invalid ($\mathsf{result} < 0$).
//Decode $\mathsf{epk}$.
$\mathsf{P1\_bytestring} \leftarrow \mathsf{epk}[0 : \mathsf{P1\_bytes}]$
$\mathsf{P2\_bytestring} \leftarrow \mathsf{epk}[\mathsf{P1\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes}]$
$\mathsf{P3\_bytestring} \leftarrow \mathsf{epk}[\mathsf{P1\_bytes} + \mathsf{P2\_bytes} : \mathsf{P1\_bytes} + \mathsf{P2\_bytes} + \mathsf{P3\_bytes}]$
$\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(1)}}(\mathsf{P1\_bytestring})$    // $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ upper triangular
$\{\mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(2)}}(\mathsf{P2\_bytestring})$    // $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
$\{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow \mathsf{Decode}_{\mathbf{P}^{(3)}}(\mathsf{P3\_bytestring})$    // $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ upper triangular

//Decode $\mathsf{sig}$.
$\mathsf{salt} \leftarrow \mathsf{sig}[\lceil nk/2\rceil : \lceil nk/2\rceil + \mathsf{salt\_bytes}]$
$\mathbf{s} \leftarrow \mathsf{Decode}_{vec}(kn, \mathsf{sig})$
for $i$ from 0 to $k-1$ do
    $\mathbf{s}_i \leftarrow \mathbf{s}[i*n : (i+1)*n]$

//Hash message and derive $\mathbf{t}$.
$\mathsf{M\_digest} \leftarrow \mathsf{SHAKE256}(\mathsf{M}, \mathsf{digest\_bytes})$    // $\mathsf{M\_digest} \in \mathcal{B}^{\mathsf{digest\_bytes}}$
$\mathbf{t} \leftarrow \mathsf{Decode}_{vec}(m, \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \lceil m\log(q)/8\rceil))$    // $\mathbf{t} \in \mathbb{F}_q^m$

//Compute $\mathcal{P}^*(\mathbf{s})$.
$\mathbf{y} \leftarrow \mathbf{0}_m$    // $\mathbf{y} \in \mathbb{F}_q^m$
$\ell \leftarrow 0$
for $i$ from 0 to $k-1$ do
    for $j$ from $k-1$ to $i$ do
        $\mathbf{u} \leftarrow \begin{cases} \left\{\mathbf{s}_i^{\mathsf{T}} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i = j \\ \left\{\mathbf{s}_i^{\mathsf{T}} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_j + \mathbf{s}_j^{\mathsf{T}} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i \neq j \end{cases}$    // $\mathbf{u} \in \mathbb{F}_q^m$
        $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}^{\ell}\mathbf{u}$
        $\ell \leftarrow \ell + 1$

//Accept signature if $\mathbf{y} = \mathbf{t}$.
if $\mathbf{y} = \mathbf{t}$ then
    return $0$
return $-1$
```

Algorithm 10 $\mathsf{MAYO.API.sign}(\mathsf{M}, \mathsf{sk})$

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 $\mathsf{sm} \in \mathcal{B}^{\mathsf{sig\_bytes}+\mathsf{mlen}}$.

```
//Expand sk.
$\mathsf{esk} \leftarrow \mathsf{MAYO.ExpandSK}(\mathsf{pk})$

//Produce signature.
$\mathsf{sig} \leftarrow \mathsf{MAYO.Sign}(\mathsf{esk}, \mathsf{M})$

//Return signed message.
return $\mathsf{sig} \parallel \mathsf{M}$
```

Algorithm 11 $\mathsf{MAYO.API.sign\_open}(\mathsf{pk}, \mathsf{M}, \mathsf{sig})$

Input: Public key $\mathsf{pk} \in \mathcal{B}^{\mathsf{cpk\_bytes}}$

Input: Signed message $\mathsf{sm} \in \mathcal{B}^{\mathsf{smlen}}$

Output: An integer result to indicate if sm is valid ($\mathsf{result} = 0$) of invalid ($\mathsf{result} < 0$) for pk

Output: The original message $\mathsf{M} \in \mathcal{B}^{\mathsf{smlen}-\mathsf{sig\_bytes}}$ if sm is valid.

```
//Expand pk.
$\mathsf{epk} \leftarrow \mathsf{MAYO.ExpandPK}(\mathsf{pk})$

//Parse signed message.
$\mathsf{sig} \leftarrow \mathsf{sm}[0 : \mathsf{sig\_bytes}]$
$\mathsf{M} \leftarrow \mathsf{sm}[\mathsf{sig\_bytes} : \mathsf{smlen}]$

//Verify signature.
$\mathsf{result} \leftarrow \mathsf{MAYO.Verify}(\mathsf{epk}, M, \mathsf{sig})$

//Return result and message.
if $\mathsf{result} < 0$ then
    $\mathsf{M} \leftarrow \perp$
return $(\mathsf{result}, \mathsf{M})$
```

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	66	78	99	133
m	64	64	96	128
o	8	18	10	12
k	9	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
$f(z)$	$f_{64}(z)$	$f_{64}(z)$	$f_{96}(z)$	$f_{128}(z)$
secret key size	24 B	24 B	32 B	40 B
public key size	1168 B	5488 B	2656 B	5008 B
signature size	321 B	180 B	577 B	838 B
expanded sk size	69 KB	92 KB	230 KB	553 KB
expanded pk size	70 KB	97 KB	233 KB	557 KB

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 three irreducible polynomials in $\mathbb{F}_{16}[z]$. (the first one is used both in MAYO$_1$ and MAYO$_2$):

$$\begin{array}{rcllllll} f_{64}(z) & = & z^{64} & & +x^3z^3 & +xz^2 & & +x^3 \\ f_{96}(z) & = & z^{96} & & +xz^3 & & +xz & +x \\ f_{128}(z) & = & z^{128} & +xz^4 & +x^2z^3 & & +x^3z & +x^2 \end{array}$$

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. Our implementations only cover parameter sets where the value of the m parameter set is divisible by 32, which includes most, but not all of the additional parameter sets. The m-values not divisible by 32 are shown in blue.

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	(66, 65, 7, 11, 16)	926 B	387 B	70 KB	71 KB
	(66, 64, 8, 9, 16)	1168 B	321 B	68 KB	70 KB
	(67, 64, 9, 8, 16)	1456 B	292 B	70 KB	72 KB
	(69, 64, 11, 7, 16)	2128 B	265 B	74 KB	76 KB
	(70, 64, 12, 6, 16)	2512 B	234 B	76 KB	78 KB
	(73, 64, 15, 5, 16)	3856 B	206 B	81 KB	85 KB
	(76, 64, 18, 4, 16)	5488 B	176 B	87 KB	92 KB
	(82, 64, 24, 3, 16)	9616 B	147 B	97 KB	107 KB
	(106, 64, 36, 2, 16)	21 KB	130 B	157 KB	178 KB
3	(99, 97, 9, 12, 16)	2198 B	626 B	233 KB	235 KB
	(99, 96, 10, 11, 16)	2656 B	576 B	230 KB	233 KB
	(100, 96, 11, 10, 16)	3184 B	532 B	234 KB	237 KB
	(101, 96, 12, 9, 16)	3760 B	486 B	238 KB	242 KB
	(102, 96, 13, 8, 16)	4384 B	440 B	242 KB	247 KB
	(104, 96, 15, 7, 16)	5776 B	396 B	251 KB	256 KB
	(107, 96, 18, 6, 16)	8224 B	353 B	263 KB	271 KB
	(110, 96, 21, 5, 16)	11 KB	307 B	276 KB	287 KB
	(115, 96, 26, 4, 16)	17 KB	262 B	297 KB	313 KB
	(124, 96, 35, 3, 16)	30 KB	218 B	334 KB	364 KB
	(156, 96, 52, 2, 16)	65 KB	188 B	510 KB	575 KB
5	(131, 130, 10, 14, 16)	3591 B	957 B	546 KB	549 KB
	(132, 129, 11, 13, 16)	4273 B	898 B	549 KB	553 KB
	(133,128, 12, 12, 16)	5008 B	838 B	553 KB	557 KB
	(134,128, 13, 11, 16)	5840 B	777 B	560 KB	566 KB
	(135,128, 14, 10, 16)	6736 B	715 B	568 KB	574 KB
	(137,128, 16, 9, 16)	8720 B	656 B	583 KB	591 KB
	(138,128, 17, 8, 16)	9808 B	592 B	590 KB	600 KB
	(141,128, 20, 7, 16)	14 KB	533 B	613 KB	626 KB
	(144,128, 23, 6, 16)	18 KB	472 B	636 KB	653 KB
	(149,128, 28, 5, 16)	26 KB	412 B	674 KB	699 KB
	(155,128, 34, 4, 16)	38 KB	350 B	719 KB	756 KB
	(167,128, 46, 3, 16)	68 KB	290 B	810 KB	877 KB
	(208,128, 68, 2, 16)	147 KB	248 B	1212 KB	1359 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, simple textbook implementation written exclusively in Sage, described in Section 3.4.

The implementations 1-3 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. 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.

Matrix-matrix and matrix-vector multiplications are, in the majority of the cases, implemented in bitsliced representation. With m being a multiple of 32, we can perform 32 parallel additions or multiplications in $\mathbb{F}_{16}$ using four 32-bit variables.

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 bit-sliced arithmetic functions are implemented for the values of m. Bit-sliced arithmetic with $m = 64$ can process 64 parallel operations using four 64-bit variables; with $m = 96$, we can process 96 parallel operations using twelve 32-bit variables; with $m = 128$, we can process 128 parallel operations using eight 64-bit variables.

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. Optimizing bit-sliced arithmetic uses a slightly different strategies for each value of m. For $m = 64$, 64 parallel multiplications are performed on data represented in one 256-bit vector register using vector permute and shuffle instructions. For $m = 96$, 96 parallel multiplications are performed on data represented in three 128-bit vector registers. For $m = 128$, 128 parallel multiplications are performed on data represented in two 256-bit registers. Further optimizations are applied by unrolling the matrix multiplication loops for each operation involved in *MAYO*, interleaving several bit-sliced arithmetic operations and by re-using intermediate values.

The echelon computation is optimized using AVX2 shuffle instructions for more efficient multiplications.

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 leads to a speed-up factor between approx. 2x-6x in KeyGen, 2x-5x in Signing (+ExpandSK) and 1.3x-2.4x in Verifying (+ExpandPK).

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 12.0.0-3ubuntu1 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$_1$	2,964,948	3,865,364	1,526,032	6,787,356	2,996,968
MAYO$_2$	6,348,792	7,512,512	2,031,976	9,290,400	2,813,708
MAYO$_3$	10,670,888	14,403,980	5,166,728	23,816,456	9,619,732
MAYO$_5$	27,467,616	38,061,916	12,344,572	59,571,696	21,619,600
Optimized Implementation		C code, using AES-NI (1st row), no AES-NI (2nd row)			
MAYO$_1$	515,168 1,978,448	766,324 2,231,480	63,448 1,526,420	1,947,392 3,415,896	397,464 1,858,496
MAYO$_2$	1,444,244 3,417,724	1,670,148 3,637,624	85,380 2,054,972	2,505,584 4,470,152	212,264 2,172,764
MAYO$_3$	4,314,644 9,280,488	7,057,268 12,013,332	213,712 5,141,692	13,179,744 18,136,112	1,982,160 6,919,956
MAYO$_5$	6,096,148 17,965,956	10,282,620 21,846,212	512,600 12,374,164	19,609,280 31,158,140	2,705,800 14,566,548
AVX2 Optimized Implementation			AVX2 compiler intrinsics and using AES-NI		
MAYO$_1$	184,116	237,424	63,356	652,052	283,228
MAYO$_2$	487,032	465,908	85,444	789,020	178,276
MAYO$_3$	956,696	1,594,572	213,032	3,249,120	1,160,552
MAYO$_5$	2,153,280	3,654,404	512,824	6,606,208	1,984,424

The fastest results on the 2.0 GHz Ice Lake platform perform KeyGen in 55 µs, Signing (+ExpandSK) in 246 µs, and Verifying (+ExpandPK) in 77 µs with MAYO$_1$. Batch signing (without expandSK) is fastest with 126 µs and MAYO$_2$. Batch verification (without expandPK) is fastest with 30 µs and MAYO$_2$.

3.4 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`.

3.5 Arm Cortex-M4 implementation

We are working on an Arm Cortex-M4 implementation based on the techniques described in [CKY21]. Preliminary results are shown in Table 3.4. We will publish an implementation supporting all parameter sets under the same license as the remaining *MAYO* code soon. We use the ST NUCLEO-L4R5ZI development board which comes with 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

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$_1$	2,579,188	3,241,021	1,539,173	5,510,453	2,703,716
MAYO$_2$	5,212,735	6,098,160	2,068,265	7,455,052	2,720,520
MAYO$_3$	9,429,436	12,431,116	5,185,043	19,672,365	8,727,014
MAYO$_5$	24,584,060	33,189,057	12,488,714	50,208,488	20,593,825
Optimized Implementation					C code, using AES-NI (1st row), no AES-NI (2nd row)
MAYO$_1$	313,438	466,279	43,830	1,496,786	283,366
	1,816,787	1,970,163	1,546,951	3,002,270	1,787,775
MAYO$_2$	921,052	1,107,113	59,130	1,826,460	159,714
	2,932,189	3,108,004	2,058,621	3,838,513	2,742,116
MAYO$_3$	4,016,744	6,566,440	147,589	12,324,612	1,637,394
	9,077,007	11,635,068	5,185,078	17,385,631	6,714,363
MAYO$_5$	4,465,717	6,764,297	355,032	16,203,574	2,244,080
	16,599,845	19,193,180	12,489,543	28,196,576	14,463,564
AVX2 Optimized Implementation					AVX2 compiler intrinsics and using AES-NI
MAYO$_1$	155,568	207,497	43,832	584,906	208,973
MAYO$_2$	419,778	427,100	59,129	697,946	129,863
MAYO$_3$	831,339	1,347,527	147,733	2,804,104	904,918
MAYO$_5$	1,727,943	2,624,093	355,030	5,148,078	1,478,483

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 12.0.1-19ubuntu3. 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	2,507,492	3,350,884	1,373,774	5,569,740	2,472,986
MAYO_2	5,256,656	6,116,216	1,836,172	7,494,964	2,424,594
MAYO_3	9,063,206	11,708,292	4,628,324	19,164,488	9,687,716
MAYO_5	23,216,914	31,509,636	11,085,626	48,646,392	18,818,812
Optimized Implementation	C code, using AES-NI (1st row), no AES-NI (2nd row)				
MAYO_1	222,666	295,314	22,392	1,087,794	205,692
	1,581,826	1,655,868	1,367,922	2,446,282	1,568,260
MAYO_2	613,636	695,602	30,642	1,269,250	118,534
	2,427,978	2,503,810	1,838,024	3,074,318	1,938,786
MAYO_3	2,917,294	4,682,590	76,164	8,839,058	1,470,684
	7,489,044	9,266,120	4,635,632	13,391,284	6,057,506
MAYO_5	4,263,490	7,038,500	180,620	13,928,986	1,783,626
	15,343,002	18,150,200	11,142,462	25,033,320	12,876,828
AVX2 Optimized Implementation	AVX2 compiler intrinsics and using AES-NI				
MAYO_1	110,112	161,988	22,518	460,978	175,158
MAYO_2	309,422	342,270	30,314	563,900	91,512
MAYO_3	508,608	633,954	75,038	1,663,666	610,010
MAYO_5	1,210,154	2,139,058	180,744	4,149,954	1,186,132

Table 3.4: *MAYO* performance in CPU cycles on the **Arm Cortex-M4** (STM32L4R5ZI). The library was compiled with the Arm GNU toolchain (`arm-none-eabi-gcc` 12.2.1). Results are the average of 1000 benchmark runs.

Scheme	`KeyGen`	`ExpandSK`	`ExpandPK`	`ExpandSK` + `Sign`	`ExpandPK` + `Verify`
MAYO$_1$	5,245,606	5,293,826	3,098,812	9,183,088	4,886,583
MAYO$_2$	11,925,130	9,418,745	4,149,236	12,033,879	5,103,238

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 12.2.1).

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 $\begin{pmatrix}\mathbf{O}^\top & \mathbf{I}_o\end{pmatrix}$. 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 $\{\mathbf{E}_{ij}\}_{1\le i\le j\le k} \in \mathbb{F}_{q^m}$, 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*

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

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

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

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:*

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

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*

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

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 the theorem 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 $T + O((Q_h + Q_s)poly(n, m, k, q))$ with:*

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

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 $(M, \mathbf{t}, \star) \in L$; 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 salt $\leftarrow$ $\mathsf{SHAKE256}(\mathsf{M\_digest} \parallel R \parallel \mathsf{seed}'_{\mathsf{sk}}, \mathsf{salt\_bytes})$. $\mathcal{B}$ aborts if there is an existing entry $(\mathsf{M\_digest}||\mathsf{salt}, \mathbf{t}, \perp)$ in L. If there is an entry $(\mathsf{M\_digest} \parallel \mathsf{salt}, \mathbf{t}, \mathbf{s})$ in L, then $\mathcal{B}$ answers with the signature $\mathsf{Encode}_{vec}(\mathbf{s}) \parallel \mathsf{salt}$. 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 $(\mathsf{Encode}_{vec}(\mathbf{s}) \parallel \mathsf{salt})$. 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 $\Pr[\mathsf{Game}_0() = 1] = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{n,m,o,k,q}(\mathcal{A})$.

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 $M \parallel R \parallel \mathsf{seed}_{\mathsf{sk}}$ or $M \parallel R \parallel \mathsf{seed}'_{\mathsf{sk}}$, 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 $(M, \mathbf{t}, \star)$ 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 $(M, \mathbf{t}, \perp)$ 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 $(\mathsf{M\_digest} \parallel \mathsf{salt}, \mathbf{t}, \perp)$ 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 $(\mathsf{M\_digest} \parallel \mathsf{salt}, \mathbf{t}, \mathbf{s})$ in L, and outputs $\mathsf{Encode}_{vec}(\mathbf{s}) \parallel \mathsf{salt}$.

   Since there are at most $Q_h + Q_s$ entries of the form $(M, \mathbf{t}, \perp)$ 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 $M \parallel \mathsf{salt} \parallel \mathsf{seed}_{\mathsf{sk}} \parallel \mathsf{ctr}$, whereas in Game_3 they are chosen at random. If the adversary does not make a random oracle query of the form $M \parallel \mathsf{salt} \parallel \mathsf{seed}_{\mathsf{sk}} \parallel \mathsf{ctr}$ (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 $(\mathsf{M}, \mathsf{sig})$ 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 $\parallel$ salt, where M_digest and salt were the message digest and salt used in one of the queries to the signing oracle. A forgery $(\mathsf{M}, \mathsf{sig})$ can only be valid for Game_4 but not for EUF-KOA game if $\mathsf{SHAKE256(M)} = \mathsf{SHAKE256(M')}$, where M′ was one of the messages queries for the signing oracle. Moreover, we must have $\mathsf{M} \neq \mathsf{M}'$, because otherwise the forgery $(\mathsf{M}, \mathsf{sig})$ is not considered valid for Game_4, so $(\mathsf{M}, \mathsf{M}')$ 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 $T + O((1 + Q_h)\mathit{poly}(n, m, k, q))$ 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:

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

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 $(M, (\mathsf{salt}, \mathbf{s}))$, 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 24 of MAYO.Sign. This factor stems from the fact that the rejection sampling functionality introduces a small amount of information 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 and MAYO_2, we choose parameters such that $m \leq ko - 8$, such that B is approximately $q^{-9} = 2^{-36}$, which means that as long as the adversary sees fewer than 2^{35} signatures, the leakage provably does not degrade security much. We expect the *MAYO* signature scheme to remain secure even if the adversary has access to an unbounded number of signatures. For MAYO_3 and MAYO_5, the value of B is approximately 2^{-60}, and 2^{-68} respectively.

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 nonhomogeneous system of m multivariate equations in n variables over $\mathbb{F}_q$ using the hybrid Wiedemann XL algorithm [BFP09, YCBC07] 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.

For underdetermined systems (i. e., $n > m$), the best approach is due to Furue, Nakamura, and Takagi [FNT21], which combines the hybrid approach with the work of Thomae and Wolf [TW12]. Their approach first reduces the underdetermined system to a set of q^k smaller overdetermined systems

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 attacks that reduce to the hybrid XL algorithm, we report the operating degree D and the optimal number of guesses k.

Parameter set	Kipnis-	Reconciliation	Intersection	Direct attack	Claw-
(n, m, o, k, q)	Shamir	(D, k)	(D, k)	(D, k)	finding
MAYO_1	222	143	255	145	143
$(66, 64, 8, 9, 16)$		(14, 9)	(7, 1)	(10, 16)	
MAYO_2	191	151	202	158	143
$(78, 64, 18, 4, 16)$		(14, 11)	(10, 0)	(12,16)	
MAYO_3	340	209	390	210	207
$(99, 96, 10, 11, 16)$		(23, 10)	(10,0)	(16,21)	
MAYO_5	461	276	525	275	272
$(133, 128, 12, 12, 16)$		(27, 18)	(12, 0)	(22, 26)	

with $m' = m - \left\lfloor \frac{n-k}{m-k} \right\rfloor - 1$ equations and $n' = m' - k$ variables, and then apply the Wiedemann XL algorithm to solve those systems. The total cost of the algorithm for $n > m$ can therefore be estimated as:

$$\mathsf{XL\_Cost}_{n,m,q} = \min_{k} q^k \cdot \mathsf{XL\_Cost}_{n',m',q}\,.$$

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 loose lower bound for the cost of a realistic attack. For example, as shown in Table 5.1, the attack with the lowest estimated bit cost against MAYO_1 is the reconciliation attack, for which the bit cost of the multiplications is estimated to be approximately 2^{143}. However, the bottleneck of the attack is computing iterated matrix-vector multiplications $\mathbf{x}_{i+1} = M\mathbf{x}_i$, where the vectors $\mathbf{x}_i$ are approximately 2^{45} elements long, and where M is very sparse. This means that storing a vector requires 17.6 Terrabytes, and the cost of accessing the entries of the vector to perform the matrix-vector multiplication is likely to significantly outweigh the cost of the multiplications themselves.

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 $\mathsf{SHAKE256}(M_1) = \mathsf{SHAKE256}(M_2)$, 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 $\mathsf{seed}_{\mathsf{sk}}$. 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 $\mathsf{sk\_seed\_bytes} = 24$, 32, or 40 for the parameter sets targeting security levels 1, 3, and 5 respectively. The bit length of $\mathsf{seed}_{\mathsf{sk}}$ 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, the majority of the zeros of $\mathcal{P}$ are 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 high 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 $o = m$.

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 $\in \mathcal{B}^{\mathsf{salt\_bytes}}$, 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 $\mathcal{P}^*(\mathbf{s}) = H$ to find a solution more efficiently compared to running a generic algorithm. Therefore, we estimate the cost of a direct attack as $\mathsf{XL\_Cost}_{kn,m,q}$.

Claw finding attacks. An attacker can compute $\mathcal{P}(\mathbf{s}_i)$ for X arbitrary inputs $\{\mathbf{s}_i\}_{i\in[X]}$ and compute $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt}_j)$ for Y arbitrary salts $\{\mathsf{salt}_j\}_{j\in[Y]}$. If $XY = q^m$, then there is a collision $\mathcal{P}(\mathbf{s}_i) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt}_j)$ with probability $\approx 1 - e^{-1}$, and the attacker can output the signature $(\mathsf{salt}_j, \mathbf{s}_i)$ for the message M. The bit-cost of the attack is then:

$$36mX + Y2^{17}\,,$$

which is equal to $12\sqrt{q^m m 2^{17}}$ for optimally chosen X, Y such that $XY = q^m$. This is the formula we use in Table 5.1. We have used gray-code enumeration [BCC+10] to evaluate $\mathcal{P}$ at X inputs and, for the sake of concreteness, we estimate that computing SHAKE256 has a bit cost of at least 2^{17}. Realistically, an attacker would use a memoryless claw finding algorithm [vW96], where it might not be possible to take full advantage of gray-code enumeration.

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 MAYO_1/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. Our generic C implementation of *MAYO* is slower than the fastest (platform-specific) optimized implementations of lattice-based signatures by only a small factor. We hope this gap will shrink as more optimized implementations of *MAYO* are developed.

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 which ignores how well attacks parallelize. Therefore, in realistic models, the state-of-the-art attacks against *MAYO* are more costly than key-search attacks on AES (which define the NIST security levels 1,3, and 5) by a wide margin.

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 40% of that of the Dilithium scheme, at security level 5, the combined size of *MAYO* is already 81% 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, Heidelberg, August 2010.

[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, Heidelberg, 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, Heidelberg, 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.

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

[DCP+20] Jintai Ding, Ming-Shing Chen, Albrecht Petzoldt, Dieter Schmidt, Bo-Yin Yang, Matthias Kannwischer, and Jacques Patarin. Rainbow. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/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 08*, volume 5037 of *LNCS*, pages 242–257. Springer, Heidelberg, June 2008.

[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, Heidelberg, 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, Heidelberg, 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, 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, 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.

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

[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, Heidelberg, May 2012.

[vW96] 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, 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.