Round 3 Version

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

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

Chapter 1

Introduction and design rationale

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

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

In the *Oil and Vinegar* scheme, the public key represents a trapdoored homogeneous multivariate map $\mathcal{P}(\mathbf{x}) = (p_0, \ldots, p_{m-1}) : \mathbb{F}_q^n \to \mathbb{F}_q^m$ which consists of a sequence of m multivariate quadratic polynomials $p_0(\mathbf{x}), \cdots, p_{m-1}(\mathbf{x})$ in n variables $\mathbf{x} = (x_0, \cdots, x_{n-1})$. The trapdoor information is a secret subspace $O \subset \mathbb{F}_q^n$ of dimension m, on which $\mathcal{P}(\mathbf{x})$ evaluates to zero. Given a salted hash digest $\mathbf{t} \in \mathbb{F}_q^m$ of a message M, the trapdoor information allows sampling a signature $\mathbf{s}$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$.

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

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

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

It is possible to add an arbitrary linear map $\mathcal{L} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ to $\mathcal{P}$ and still sample preimages efficiently using the same strategy:

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

Kipnis, Patarin, and Goubin proposed in the UOV paper [KPG99] to choose the linear map $\mathcal{L}$ based on a hash of the salted message, so that the trapdoor function is different for each signature. The purpose of this variant is to prevent claw-finding attacks, and is essentially free.

One practical drawback of the Oil and Vinegar 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 $m^3/2$ coefficients 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 $\mathcal{P}^\star(\mathbf{x}_0, \ldots, \mathbf{x}_{k-1}) : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$, where k is a parameter of the scheme. The whipped map $\mathcal{P}^\star$ is constructed in such a way that it evaluates to zero on the subspace $O^k = \{(\mathbf{o}_0, \ldots, \mathbf{o}_{k-1}) \,|\, \forall i : \mathbf{o}_i \in O\}$ which has dimension ko. Concretely, we define:

$$\mathcal{P}^\star(\mathbf{x}_0, \ldots, \mathbf{x}_{k-1}) := \sum_{i=0}^{k-1} \mathbf{E}_{ii}\mathcal{P}(\mathbf{x}_i) + \sum_{i=0}^{k-1}\sum_{j=i+1}^{k-1} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{x}_i, \mathbf{x}_j)$$

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 $ko \geq m$ to make sure that the space O^k is large enough so that the signer can sample signatures $\mathbf{s} = (\mathbf{s}_0, \cdots, \mathbf{s}_{k-1})$ such that $\mathcal{P}^\star(\mathbf{s}) = \mathbf{t}$ with the usual *Oil and Vinegar* approach. The signer first samples $(\mathbf{v}_0, \ldots, \mathbf{v}_{k-1}) \in \mathbb{F}_q^{kn}$ at random, and then solves for $(\mathbf{o}_0, \ldots, \mathbf{o}_{k-1}) \in O^k$ such that

$$\mathcal{P}^\star(\mathbf{v}_0 + \mathbf{o}_0, \ldots, \mathbf{v}_{k-1} + \mathbf{o}_{k-1}) + \mathcal{L}(\mathbf{v}_0 + \mathbf{o}_0, \ldots, \mathbf{v}_{k-1} + \mathbf{o}_{k-1}) = \mathbf{t}$$

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 $(n, m, o, k, q) = (86, 64, 13, 5, 16)$, which results in a public key of just 2928 bytes, and a signature size of 239 bytes (215 bytes for $\mathbf{s}$ and 24 bytes for the salt). Compared to the compressed *Oil and Vinegar* scheme at the same security level (66 KB), this is a 23-fold reduction in public key size at the cost of a 2.5-fold increase in signature size. We also propose a parameter set with even smaller public keys, but larger signatures.

Adding linear terms. The choice to include the linear terms $\mathcal{L}$ into the trapdoor function is non-standard. The purpose of this is to prevent claw-finding attacks. Without the term, one can compute $\mathcal{P}^\star(\mathbf{s})$ for many candidate signatures $\mathbf{s}$ and $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt})$ for many values of (M, salt) until a collision is found, which yields a forged signature with complexity $O(q^{m/2})$. The linear term $\mathcal{L}(\mathbf{s})$ prevents this attack, since it depends both on $\mathbf{s}$ and (M,salt). Concretely, we define $\mathcal{L}(\mathbf{s})$ as

$$\mathcal{L}(\mathbf{\Lambda}, \mathbf{s}_0, \ldots, \mathbf{s}_{k-1}) := \sum_{i=0}^{k-1} \mathbf{E}_{i0} \cdot \mathbf{\Lambda} \cdot \mathbf{s}_i \,,$$

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

where $\mathbf{\Lambda} \in \mathbb{F}_q^{m \times n}$ is sampled using AES-CTR from a 16-byte AES-128 key and a 16-byte nonce, which are derived from the hash of $\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt}$. It can be checked that this is a universal hashing family, using the fact that all non-trivial linear combinations of the $\mathbf{E}_{i0}$ matrices have full rank. We choose this family of linear maps because it is compatible with the whipping structure of $\mathcal{P}^\star$, which allows us to avoid any computational overhead from the contributions of $\mathcal{L}$ to the signing and verification. As it turns out, we just need to expand $\mathbf{\Lambda}$ from a seed and XOR it into some intermediate values during signing and verification. Therefore, we get the security benefits of the linear term $\mathcal{L}$ without any meaningful performance penalty.

Choice of finite field. We choose to instantiate MAYO over $\mathbb{F}_{16}$, the finite field of order 16. Choosing a single field for all the parameter sets simplifies implementation and allows for code reuse to reduce code size. Arithmetic over $\mathbb{F}_{16}$ can be implemented very efficiently on many platforms using either shuffle-based instructions or elementary operations such as XOR and bit shifts. Since $\mathbb{F}_{16}$ is a field of characteristic 2, it is trivial to store and randomly sample field elements, and the various MAYO algorithms can be expressed easily as binary circuits. This makes characteristic 2 especially suitable for use in MPC protocols, zero-knowledge proofs, or masked implementations. A drawback of fields of characteristic 2 is that the wedge key-recovery attack [Ran26] appears to be more efficient in characteristic 2 than in odd characteristic. However, the wedge attack and its variants [FI26] can be avoided rather cheaply by properly choosing parameters, even in characteristic 2. Therefore, we choose to retain the aforementioned benefits of characteristic 2, and account for the wedge attack in our parameter selection.

The use of a salt. A signature for a message M is a pair $(\mathbf{s}, \mathsf{salt})$ such that $\mathcal{P}^*(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$, where $\mathbf{t}$ and $\mathbf{\Lambda}$ are derived from $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt})$. It is possible to prove the security of MAYO without the salt (under the same assumptions and limitations as our security proof), but only when signing is deterministic. The main purpose of the salt is therefore to allow for randomized signing, which helps protect an implementation of MAYO against side-channel attacks and fault injection attacks.

Remark. *In MAYO, when a signing attempt fails, we keep the same salt and retry with new randomness for the preimage sampling subroutine. This is in contrast to some other multivariate signature schemes such as QR-UOV [AFI+24], which instead keep the randomness for the preimage sampling subroutine and keep sampling new* salt *values until a preimage is found. The advantage of this alternative approach is that it allows for a security proof whose loss does not depend on the number of signatures seen by an adversary [KX24]. However, there are no known attacks that are prevented by this alternative approach, and implementing it in a constant-time manner is extremely expensive.*[2] *Therefore we choose not to adopt this approach.*

Change Log

Modifications in Third Round Submission.

- **Updated parameters for MAYO_1 and MAYO_2.** We decided to make a minor change to MAYO_1 for the round-3 submission to slightly increase the security against a direct attack, at essentially no cost in performance or sizes (2% increase). In response to the wedge attack which was discovered in the second round we modified the MAYO_2 parameter set to increase the cost of key-recovery attacks.[3] This change resulted in a 28% increase in signature size, a 40% reduction in key size,

[2] In a nutshell, the problem is that the expected number of signing attempts depends strongly on the choice of sampling randomness (the so-called vinegar variables), which need to be kept secret. In contrast, with the traditional signing mode (constant salt, resample fresh vinegar values on failure) this problem does not appear, since the number of signing attempts is independent of the final signature and does not need to be kept secret.

[3] The bit cost of the wedge attack against MAYO_2 was 2^{112} which is lower than the target value of 2^{143}. However, we argue that the attack is much less threatening than the 2^{112} number suggests. The attack requires doing 2^{49} *sequential* matrix-vector products with a bit cost of 2^{63} each. The sequential nature of the attack limits the amount of parallelization that can be used

and improved keygen, signing, and verification performance. Initially, we wanted to change the MAYO_2 parameters to $(n, m, o, k, q) = (82, 64, 13, 5, 16)$. However, days before the submission deadline we were notified privately by Lars Ran and Simon-Philipp Merz of a new key-recovery attack on UOV-like public keys that is more broadly applicable than the Wedge attack. We used $n = 86$ instead of $n = 82$ to push the cost of this new attack above 2^{143}. The new attack will be made public relatively soon. As we are not yet aware of the technical details of this attack, we have not included it in the security analysis section.

- **MDS whipping schedule.** We change the whipping schedule $\{\mathbf{E}_{ij}\}_{i\in[k],j\in[k]}$. These matrices correspond to multiplication by powers of z in $\mathbb{F}_q[z]/(f_m(z))$. In the second round submission the order of these matrices was $\mathbf{E}_{ij} \leftrightarrow z^{\ell(i,j)}$, where $\ell(i,j) = ik - i(i+1)/2 + j$ for $i \le j$, and $\ell(i,j) = \ell(j,i)$ otherwise. (This ℓ was implicitly defined by the order in which we loop over (i,j) in the signing algorithm.) However, with this choice, the symmetric matrix $\mathbf{Z} := \{z^{\ell(i,j)}\}_{ij}$ has submatrices of unusually low rank, which can be exploited to mildly speed up collision attacks. In the third round submission we have changed the order of the powers of z to make the whipping schedule $\mathbf{Z}$ an MDS matrix (i.e., all submatrices have full rank) to eliminate this weakness. This has no impact on the efficiency of the scheme.
- **Adding linear terms to the trapdoor map.** In the second-round submission the public trapdoor map $\mathcal{P}^\star$ is a homogeneous quadratic map. We decided to add linear terms to the verification equation, which now becomes $\mathcal{P}^\star(\mathbf{s}) + L(\mathbf{s}) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt})$, where the coefficients of the linear map L are determined from the hash of $\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt}$. The purpose of this modification is to prevent collision attacks. Since both sides of the verification equation depend on M and salt, one can no longer perform a collision search to find forgeries. This is an old idea, proposed already in the original UOV paper in 1999. We have added this modification to the third-round submission, in response to some improvements in claw-finding attacks that were communicated to us privately in August 2026.
- **Improved implementations.** We have made several improvements to our implementations of MAYO, increasing the speed of key-generation, signing, and verification, without the need for changing KAT values. We added an alternative approach to signing which avoids using the L value which is precomputed during $\mathsf{MAYO.ExpandSK}$. This is 20%–30% more efficient when signing using a compact representation of the secret key (because we don't need to compute L), but less efficient when signing a batch of messages (because L can be computed once and used for all of the signatures). We have made the new approach without L the default implementation when signing a message using the compact secret key, while keeping the old approach when signing from a pre-expanded secret key. This approach was suggested to us and implemented by Hiroshi Amagasa, who is added to the list of implementers in the third-round submission. Moreover, we have also added an implementation which uses AVX512, GFNI, and VAES instructions, which gives significant speedups on platforms where these are available.
- **Expanded design rationale.** The NIST IR 8610 second round status report mentioned it might be worthwhile to consider odd characteristic parameter sets, and the use of a "salt-UOV" variant of MAYO. We have added the "Choice of finite field." and "The use of a salt." paragraphs to section 1 to explain our design choices.
- **Updated security analysis.** We updated the security analysis section with references to the new Wedge attack and improved algorithms for finding solutions to underdetermined systems of quadratic equations. We updated the security proof to include the linear terms in the trapdoor map.

and forces a long latency of the attack. For example, even if humanity manages to build a machine that does the huge matrix-vector multiplication in 0.02 milliseconds (which is faster than an honest signing or verification operation), the attack would still take 357 years. Simply throwing more hardware at the problem does not bring down the latency. This is reminiscent of the analysis of Grover's algorithm on AES-128. Despite the low gate count, Grover's algorithm is not considered to be a threat against AES-128 in practice, because it requires a very large depth (and thus latency).

Modifications in Second Round Submission.

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

Differences between first round submission and [Beu22].

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

Chapter 2

The MAYO protocol specification

2.1 Written specification

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

2.1.1 Parameters

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

- q, the size of a finite field $\mathbb{F}_q$. In this specification, we fix $q = 16$.
- m, the number of multivariate quadratic polynomials in the public key. We choose it to be even.
- n, the number of variables in the multivariate quadratic polynomials in the public key.
- o, the dimension of the oil space O, satisfying $n - o \geq m$.
- k, the whipping parameter, satisfying $k < n - o$, $ko \geq m$, and $k(k+1)/2 \leq m$.
- 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 seed_{pk}.
- $f(z) \in \mathbb{F}_q[z]$, an irreducible polynomial of degree m, defining a field extension $K := \mathbb{F}_q[z]/(f(z))$ of degree m over $\mathbb{F}_q$.
- $\mathbf{Z}$, a symmetric $k \times k$ MDS matrix over K whose $k(k+1)/2$ upper diagonal entries are linearly independent over $\mathbb{F}_q$. (In our instantiation, the entries of $\mathbf{Z}$ are powers of z from z^0 to $z^{k(k+1)/2-1}$, although this is not a requirement.)

These parameters define the following values:

- sk_seed_bytes $=$ R_bytes $=$ salt_bytes: The length of R and of seed_{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.
- $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$, for all $i, j \in [k]$, the matrix that corresponds to multiplication by $\mathbf{Z}[i,j] \bmod f(z)$.

2.1.2 Preliminaries and notation

Notation. If X is a finite set, we write $x \xleftarrow{\$} X$ to denote that x is assigned a value chosen from X uniformly at random. If A is an algorithm, we write $x \leftarrow A(y)$ to denote that x is assigned the output of running A on input y. If k is an integer, we denote by $[k]$ the set $\{0,\ldots,k-1\}$. 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, the result of which 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. $\mathbf{AB} \in \mathbb{F}_{16}^{m\times k}$ is the matrix whose entry in row i and column j is equal to $\sum_{l=0}^{n-1} \mathbf{A}[i,l]\mathbf{B}[l,j]$. 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 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 $ko \geq m$, 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 $\mathbf{A}\mathbf{x} = \mathbf{y}$. The solution space has dimension $ko - m$, and the random vector $\mathbf{r} \in \mathbb{F}_q^{ko}$ is used to pick each of the q^{ko-m} solutions with equal probability. This is done by solving the related system $\mathbf{A}\mathbf{x}' = y - \mathbf{A}r$ with the usual Gaussian Elimination approach, and outputting $\mathbf{x} = \mathbf{x}' + \mathbf{r}$. If the input matrix $\mathbf{A}$ does not have rank m, then $\mathsf{SampleSolution}(\mathbf{A}, \mathbf{y}, \mathbf{r})$ outputs $\perp$, even in the somewhat unlikely case that the system $\mathbf{A}\mathbf{x} = \mathbf{y}$ has solutions. SampleSolution uses a subroutine EF (see Algorithm 1) that performs elementary row operations on an input matrix $\mathbf{B} \in \mathbb{F}_q^{m\times(ko+1)}$ 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 $i - 1$.

Remark. *As stated in Algorithm 1, the* EF *algorithm has secret-dependent branching to do the pivot selection. The implementer must take care to realize this algorithm in a constant-time manner, to avoid leaking information about the input matrix.*

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 | pivot_row ≤ i < m and B[i, pivot_column] ≠ 0}
    if possible_pivots = ∅ then
        pivot_column ← pivot_column + 1          // Move to next column if there is no pivot.
        continue
    next_pivot_row ← min(possible_pivots)
    Swap(B[pivot_row, :], B[next_pivot_row, :])

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

    //Eliminate entries below the pivot.
    for row from next_pivot_row + 1 to m − 1 do
        B[row, :] ← B[row, :] − B[row, pivot_column] 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 $\mathbf{x} \in \mathbb{F}_q^{ko}$ that satisfies $\mathbf{Ax} = \mathbf{y}$ if $\mathrm{rank}(\mathbf{A}) = m$; otherwise, output $\perp$.

1: //Randomize the system using $\mathbf{r}$.
2: $\mathbf{x} \leftarrow \mathbf{r} \in \mathbb{F}_q^{ko}$
3: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{Ar}$
4:
5: //Put $(\mathbf{A}\,\mathbf{y})$ in echelon form with leading 1's.
6: $(\mathbf{A}\,\mathbf{y}) \leftarrow \mathsf{EF}(\mathbf{A}\,\mathbf{y})$
7:
8: //Check if $\mathbf{A}$ has rank m.
9: **if** $\mathbf{A}[m-1, :] = \mathbf{0}_{ko}$ **then**
10: **return** $\perp$
11:
12: //Back-substitution
13: **for** r from $m-1$ to 0 **do**
14: //Let c be the index of the first non-zero element of $A[r, :]$.
15: $\mathbf{x}_c \leftarrow \mathbf{x}_c + \mathbf{y}_r$
16: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{y}_r A[:, c]$
17: **return** $\mathbf{x}$

2.1.3 Hashing and randomness expansion

SHAKE256. We use the SHAKE256 extendable-output function for the purpose of hashing and sampling secret material. We denote by $\mathsf{SHAKE256}(X, l)$ 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 $\mathcal{P}$, and to generate the coefficients of Λ. We define the function $\mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}, \mathsf{iv}, l)$, which takes a 16-byte seed seed, and produces l bytes of output. The output is the first l bytes of the concatenation of the AES-128 encryptions of the blocks $\mathsf{iv} \oplus 0, \mathsf{iv} \oplus 1, \cdots, \mathsf{iv} \oplus (\lceil l/16 \rceil - 1)$, using seed as the key in the AES-128 block cipher [AES01]. Here, the block counter i is encoded as a 16-byte big-endian integer before it is XORed with iv. We define $\mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}, l) := \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{seed}, 0^{128}, l)$. The implementation of the AES-128 block cipher does not need to be constant-time or side-channel secure, because the key, the input, iv, 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: $\mathsf{Encode}_{\mathbb{F}_{16}}(a) \in [16]$

We encode a field element $a = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ as a nibble $\mathsf{Encode}_{\mathbb{F}_{16}}(a) \in [16]$, 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: $\mathsf{Decode}_{\mathbb{F}_{16}}(\mathsf{nibble})$

The operation $\mathsf{Decode}_{\mathbb{F}_{16}}$ is the inverse of $\mathsf{Encode}_{\mathbb{F}_{16}}$. 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 $\mathsf{Encode}_{\mathbb{F}_{16}}(x_0), \ldots, \mathsf{Encode}_{\mathbb{F}_{16}}(x_{n-1})$ (the encodings of field elements with even indices go in the low nibble of their bytes), and padding with the zero nibble if n is odd.

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

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

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

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

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

The operation $\mathsf{Decode}_{\mathbf{O}}(\mathsf{bytestring})$ takes a byte-string 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 $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$.

The operation $\mathsf{Decode}_{\mathbf{\Lambda}}(\mathsf{bytestring})$ takes a byte-string of length $mn/2$ as input and outputs a matrix $\mathbf{\Lambda} \in \mathbb{F}_q^{m\times n}$ such that column i of $\mathbf{\Lambda}$ is equal to $\mathsf{Decode}_{vec}(m, \mathsf{bytestring}[i \cdot m/2 : (i+1) \cdot m/2])$ for all $i \in [n]$. By $\mathbf{\Lambda}_v, \mathbf{\Lambda}_o \leftarrow \mathsf{Decode}_{\mathbf{\Lambda}}(\mathsf{bytestring})$ we denote that $\mathbf{\Lambda}_v \in \mathbb{F}_q^{m\times(n-o)}$ receives the first $n-o$ columns of the output of $\mathsf{Decode}_{\mathbf{\Lambda}}(\mathsf{bytestring})$ and $\mathbf{\Lambda}_o \in \mathbb{F}_q^{m\times o}$ receives the remaining o columns. (Note that $\mathsf{Decode}_{\mathbf{\Lambda}}$ parses its bytestring input in a column-major order, unlike $\mathsf{Decode}_{\mathbf{O}}$ which uses a row-major order.)

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

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

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

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

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

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

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

Algorithm 3 EncodeMatrices$(r, c, \{\mathbf{A}_i\}_{i\in[m]}$, 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: is_triangular $\in \{0,1\}$, a bit to indicate if the $\mathbf{A}_i$ are upper triangular or not.
Output: A byte string bytestring $\in \mathcal{B}^{mrc/2}$ if is_triangular $=$ false, bytestring $\in \mathcal{B}^{mr(r+1)/4}$ otherwise.

```
bytestring ← ∅
for i from 0 to r − 1 do
    for j from 0 to c − 1 do
        if i ≤ j or is_triangular = false then
            bytestring ← bytestring||Encode_vec({A_s[i, j]}_{s∈[m]})
return bytestring.
```

3. $\mathsf{Encode}_{\mathbf{P}^{(3)}}(\cdot) := \mathsf{EncodeMatrices}(o, o, \cdot, \mathsf{true})$
4. $\mathsf{Encode}_{\mathbf{L}}(\cdot) := \mathsf{Encode}_{\mathbf{P}^{(2)}}(\cdot)$

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

2.1.5 The Basic MAYO functionalities

We define five functionalities:

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

Remark. *In lines 36,37, and 39 of the MAYO.Sign algorithm and line 28 of the MAYO.Verify algorithm we accumulate values of the form $\mathbf{E}_{ij}\mathbf{y}$, where $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ are matrices that represent multiplication by powers of z in a finite field $\mathbb{F}_q[z]/f(z)$. Rather than computing the matrix multiplications explicitly, it is more efficient to accumulate the values in a single polynomial and do a single reduction mod $f(z)$, or to use Horner's method.*

Remark. *Line 27 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 35 of MAYO.Sign.*

Remark. *The operation on line 33 of the MAYO.Sign algorithm can be performed implicitly by just adding $\mathbf{\Lambda}_v$ to some intermediate value during the computation of $\mathbf{u}$ on line 35, thereby reusing the existing multiplications by $\mathbf{E}_{i0}$ and $\mathbf{v}_i$. The same holds for line 25 of the MAYO.Verify algorithm.*

2.1.6 Implementing the NIST API

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

- MAYO.API.keypair (Algorithm 4): Outputs a pair $(\mathsf{sk}, \mathsf{pk}) \in \mathcal{B}^{\mathsf{csk\_bytes}} \times \mathcal{B}^{\mathsf{cpk\_bytes}}$, where sk and pk are compact representations of a MAYO secret key and public key respectively. This algorithm

Algorithm 4 MAYO.CompactKeyGen()

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

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

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

Algorithm 5 MAYO.ExpandSK(csk)

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

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

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

Algorithm 6 MAYO.ExpandPK(cpk)

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

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

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

is identical to MAYO.CompactKeyGen.

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

Algorithm 7 MAYO.Sign(esk, M)

Input: Expanded secret key $\mathsf{esk} \in \mathcal{B}^{\mathsf{esk\_bytes}}$
Input: Message $\mathsf{M} \in \mathcal{B}^*$
Constant: $\{\mathbf{E}_{ij}\}_{i,j\in[k]} \subset \mathbb{F}_q^{m\times m}$ // $\mathbf{E}_{ij}$ represents multiplication by $\mathbf{Z}[i,j]$ in $\mathbb{F}_q[z]/(f(z))$
Output: Signature $\mathsf{sig} \in \mathcal{B}^{\mathsf{sig\_bytes}}$

//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, $\mathbf{t}$, and $\mathbf{\Lambda}$.
$\mathsf{M\_digest} \leftarrow \mathsf{SHAKE256}(M, \mathsf{digest\_bytes})$ // $\mathsf{M\_digest} \in \mathcal{B}^{\mathsf{digest\_bytes}}$
$\mathsf{R} \leftarrow 0_{\mathsf{R\_bytes}}$ or $\mathsf{R} \xleftarrow{\$} \mathcal{B}^{\mathsf{R\_bytes}}$ // Optional randomization
$\mathsf{salt} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{R} \,\|\, \mathsf{seed_{sk}}, \mathsf{salt\_bytes})$ // $\mathsf{salt} \in \mathcal{B}^{\mathsf{salt\_bytes}}$
$\mathsf{hash} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{salt}, m/2 + 32)$
$\mathbf{t} \leftarrow \mathsf{Decode}_{vec}(m, \mathsf{hash}[0 : m/2])$ // $\mathbf{t} \in \mathbb{F}_q^m$
$\mathsf{Lambda\_Bytes} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{hash}[m/2 : m/2 + 16], \mathsf{hash}[m/2 + 16 : m/2 + 32], nm/2)$
$\mathbf{\Lambda}_v, \mathbf{\Lambda}_o \leftarrow \mathsf{Decode}_{\mathbf{\Lambda}}(\mathsf{Lambda\_Bytes})$ // $\mathbf{\Lambda}_v \in \mathbb{F}_q^{m\times(n-o)}$, $\mathbf{\Lambda}_o \in \mathbb{F}_q^{m\times o}$

//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 $\mathbf{Ax} = \mathbf{y}$.
  $\mathbf{A} \leftarrow 0_{m\times ko} \in \mathbb{F}_q^{m\times ko}$
  $\mathbf{y} \leftarrow \mathbf{t}$
  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 $\mathbf{M}_i$
  $\mathbf{M}_0 \leftarrow \mathbf{M}_0 + \mathbf{\Lambda}_v\mathbf{O} + \mathbf{\Lambda}_o$
  for i from 0 to $k-1$ **do**
    $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}_{i0}\mathbf{\Lambda}_v\mathbf{v}_i$
    for j from i to $k-1$ **do**
      $\mathbf{u} \leftarrow \begin{cases} \{\mathbf{v}_i^\mathsf{T}\mathbf{P}_a^{(1)}\mathbf{v}_i\}_{a\in[m]} & \text{if } i = j \\ \{\mathbf{v}_i^\mathsf{T}\mathbf{P}_a^{(1)}\mathbf{v}_j + \mathbf{v}_j^\mathsf{T}\mathbf{P}_a^{(1)}\mathbf{v}_i\}_{a\in[m]} & \text{if } i \neq j \end{cases}$ // $\mathbf{u} \in \mathbb{F}_q^m$
      $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}_{ij}\mathbf{u}$
      $\mathbf{A}[:, i*o : (i+1)*o] \leftarrow \mathbf{A}[:, i*o : (i+1)*o] + \mathbf{E}_{ij}\mathbf{M}_j$
      if $i \neq j$ **then**
        $\mathbf{A}[:, j*o : (j+1)*o] \leftarrow \mathbf{A}[:, j*o : (j+1)*o] + \mathbf{E}_{ij}\mathbf{M}_i$

  //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 8 MAYO.Verify(epk, M, 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: $\{\mathbf{E}_{ij}\}_{i,j\in[k]} \subset \mathbb{F}_q^{m\times m}$ // $\mathbf{E}_{ij}$ represents multiplication by $\mathbf{Z}[i,j]$ in $\mathbb{F}_q[z]/(f(z))$

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

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

//Hash message and derive $\mathbf{t}$ and $\mathbf{\Lambda}$.
$\mathsf{M\_digest} \leftarrow \mathsf{SHAKE256}(\mathsf{M}, \mathsf{digest\_bytes})$ // $\mathsf{M\_digest} \in \mathcal{B}^{\mathsf{digest\_bytes}}$
$\mathsf{hash} \leftarrow \mathsf{SHAKE256}(\mathsf{M\_digest} \,\|\, \mathsf{salt}, m/2+32)$
$\mathbf{t} \leftarrow \mathsf{Decode}_{vec}(m, \mathsf{hash}[0 : m/2])$ // $\mathbf{t} \in \mathbb{F}_q^m$
$\mathsf{Lambda\_Bytes} \leftarrow \mathsf{AES\text{-}128\text{-}CTR}(\mathsf{hash}[m/2 : m/2+16], \mathsf{hash}[m/2+16 : m/2+32], nm/2)$
$\mathbf{\Lambda} \leftarrow \mathsf{Decode}_{\mathbf{\Lambda}}(\mathsf{Lambda\_Bytes})$ // $\mathbf{\Lambda} \in \mathbb{F}_q^{m\times n}$

//Compute $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s})$.
$\mathbf{y} \leftarrow \mathbf{0}_m$ // $\mathbf{y} \in \mathbb{F}_q^m$
for i from 0 to $k-1$ **do**
    $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}_{i0}\mathbf{\Lambda}\mathbf{s}_i$
    for j from i to $k-1$ **do**
        $$\mathbf{u} \leftarrow \begin{cases} \left\{ \mathbf{s}_i^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i = j \\ \left\{ \mathbf{s}_i^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_j + \mathbf{s}_j^\mathsf{T} \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ \mathbf{0} & \mathbf{P}_a^{(3)} \end{pmatrix} \mathbf{s}_i \right\}_{a\in[m]} & \text{if } i \neq j \end{cases}$$ // $\mathbf{u} \in \mathbb{F}_q^m$
        $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}_{ij}\mathbf{u}$

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

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

Input: A message $\mathsf{M} \in \mathcal{B}^{\mathsf{mlen}}$

Input: Secret key $\mathsf{sk} \in \mathcal{B}^{\mathsf{csk\_bytes}}$

Output: A signed message $\mathsf{sm} \in \mathcal{B}^{\mathsf{sig\_bytes}+\mathsf{mlen}}$.

```
//Expand sk.
esk ← MAYO.ExpandSK(sk)

//Produce signature.
sig ← MAYO.Sign(esk, M)

//Return signed message.
return sig ‖ M
```

Algorithm 10 MAYO.API.sign_open(pk, sm)

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 (result $= 0$) or invalid (result < 0) for pk

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

```
//Expand pk.
epk ← MAYO.ExpandPK(pk)

//Parse signed message.
sig ← sm[0 : sig_bytes]
M ← sm[sig_bytes : smlen]

//Verify signature.
result ← MAYO.Verify(epk, M, sig)

//Return result and message.
if result < 0 then
    M ← ⊥
return (result, M)
```

2.1.7 Parameter sets

2.1.7.1 Chosen parameter sets.

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

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

$$\begin{array}{rcllll} f_{64}(z) & = & z^{64} & +x^3z^3 & +xz^2 & +x^3 \\ f_{80}(z) & = & z^{80} & +xz^3 & +x^2z^2 & +x \\ f_{108}(z) & = & z^{108} & +(x^2+x+1)z^3 & +z^2 & +x^3 \\ f_{142}(z) & = & z^{142} & +z^3 & +x^3z^2 & +x^2 \end{array}$$

We use the following whipping schedules. They are determined by sampling random permutations of the powers of z by hashing "MAYO_{k}_{trial}", for trial counting up from 0, until the MDS condition is satisfied. The script that was used is included in the submission package.

$$\mathbf{Z}_5 = \begin{pmatrix} z^{11} & z^{13} & z^3 & z^4 & z^6 \\ z^{13} & z^0 & z^{14} & z^{10} & z^2 \\ z^3 & z^{14} & z^8 & z^1 & z^5 \\ z^4 & z^{10} & z^1 & z^{12} & z^9 \\ z^6 & z^2 & z^5 & z^9 & z^7 \end{pmatrix}, \qquad \mathbf{Z}_{10} = \begin{pmatrix} z^{10} & z^{25} & z^7 & z^{44} & z^{40} & z^{24} & z^{12} & z^{32} & z^5 & z^{27} \\ z^{25} & z^6 & z^{45} & z^{50} & z^{19} & z^0 & z^{11} & z^{14} & z^{37} & z^{52} \\ z^7 & z^{45} & z^{51} & z^{16} & z^{42} & z^{30} & z^{53} & z^8 & z^{21} & z^4 \\ z^{44} & z^{50} & z^{16} & z^{15} & z^{39} & z^{26} & z^{48} & z^{54} & z^{23} & z^{34} \\ z^{40} & z^{19} & z^{42} & z^{39} & z^{22} & z^{33} & z^1 & z^{35} & z^{41} & z^{18} \\ z^{24} & z^0 & z^{30} & z^{26} & z^{33} & z^{13} & z^{46} & z^{17} & z^2 & z^{47} \\ z^{12} & z^{11} & z^{53} & z^{48} & z^1 & z^{46} & z^{36} & z^{20} & z^{49} & z^{43} \\ z^{32} & z^{14} & z^8 & z^{54} & z^{35} & z^{17} & z^{20} & z^{38} & z^{29} & z^9 \\ z^5 & z^{37} & z^{21} & z^{23} & z^{41} & z^2 & z^{49} & z^{29} & z^3 & z^{31} \\ z^{27} & z^{52} & z^4 & z^{34} & z^{18} & z^{47} & z^{43} & z^9 & z^{31} & z^{28} \end{pmatrix},$$

$$\mathbf{Z}_{11} = \begin{pmatrix} z^{35} & z^{13} & z^{39} & z^{32} & z^4 & z^7 & z^{40} & z^{14} & z^{43} & z^{62} & z^{64} \\ z^{13} & z^{55} & z^{42} & z^{11} & z^{63} & z^{19} & z^{58} & z^{49} & z^{36} & z^{12} & z^{20} \\ z^{39} & z^{42} & z^6 & z^{15} & z^{57} & z^{47} & z^{16} & z^{34} & z^{44} & z^{46} & z^{50} \\ z^{32} & z^{11} & z^{15} & z^{18} & z^5 & z^{21} & z^{28} & z^{10} & z^{51} & z^{24} & z^{31} \\ z^4 & z^{63} & z^{57} & z^5 & z^{38} & z^0 & z^{56} & z^{29} & z^{54} & z^{60} & z^{23} \\ z^7 & z^{19} & z^{47} & z^{21} & z^0 & z^{22} & z^{27} & z^{61} & z^3 & z^{48} & z^8 \\ z^{40} & z^{58} & z^{16} & z^{28} & z^{56} & z^{27} & z^{37} & z^{45} & z^{30} & z^1 & z^{17} \\ z^{14} & z^{49} & z^{34} & z^{10} & z^{29} & z^{61} & z^{45} & z^{26} & z^{52} & z^{59} & z^{41} \\ z^{43} & z^{36} & z^{44} & z^{51} & z^{54} & z^3 & z^{30} & z^{52} & z^{25} & z^9 & z^2 \\ z^{62} & z^{12} & z^{46} & z^{24} & z^{60} & z^{48} & z^1 & z^{59} & z^9 & z^{53} & z^{65} \\ z^{64} & z^{20} & z^{50} & z^{31} & z^{23} & z^8 & z^{17} & z^{41} & z^2 & z^{65} & z^{33} \end{pmatrix},$$

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	88	86	118	154
m	80	64	108	142
o	8	13	10	12
k	10	5	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
$f(z)$	$f_{80}(z)$	$f_{64}(z)$	$f_{108}(z)$	$f_{142}(z)$
$\mathbf{Z}$	$\mathbf{Z}_{10}$	$\mathbf{Z}_5$	$\mathbf{Z}_{11}$	$\mathbf{Z}_{12}$
secret key size	24 B	24 B	32 B	40 B
public key size	1456 B	2928 B	2986 B	5554 B
signature size	464 B	239 B	681 B	964 B
expanded sk size	152 KB	115 KB	368 KB	823 KB
expanded pk size	153 KB	117 KB	371 KB	828 KB

$$\mathbf{Z}_{12} = \begin{pmatrix} z^{62} & z^{30} & z^{52} & z^{53} & z^{24} & z^{0} & z^{25} & z^{12} & z^{71} & z^{20} & z^{57} & z^{32} \\ z^{30} & z^{7} & z^{37} & z^{55} & z^{43} & z^{70} & z^{31} & z^{47} & z^{35} & z^{9} & z^{69} & z^{15} \\ z^{52} & z^{37} & z^{64} & z^{26} & z^{76} & z^{45} & z^{54} & z^{10} & z^{40} & z^{34} & z^{11} & z^{19} \\ z^{53} & z^{55} & z^{26} & z^{61} & z^{16} & z^{60} & z^{5} & z^{18} & z^{42} & z^{17} & z^{8} & z^{75} \\ z^{24} & z^{43} & z^{76} & z^{16} & z^{41} & z^{27} & z^{6} & z^{2} & z^{4} & z^{51} & z^{68} & z^{38} \\ z^{0} & z^{70} & z^{45} & z^{60} & z^{27} & z^{63} & z^{22} & z^{49} & z^{23} & z^{1} & z^{21} & z^{48} \\ z^{25} & z^{31} & z^{54} & z^{5} & z^{6} & z^{22} & z^{14} & z^{29} & z^{44} & z^{65} & z^{77} & z^{72} \\ z^{12} & z^{47} & z^{10} & z^{18} & z^{2} & z^{49} & z^{29} & z^{58} & z^{67} & z^{59} & z^{39} & z^{56} \\ z^{71} & z^{35} & z^{40} & z^{42} & z^{4} & z^{23} & z^{44} & z^{67} & z^{33} & z^{73} & z^{36} & z^{74} \\ z^{20} & z^{9} & z^{34} & z^{17} & z^{51} & z^{1} & z^{65} & z^{59} & z^{73} & z^{28} & z^{46} & z^{3} \\ z^{57} & z^{69} & z^{11} & z^{8} & z^{68} & z^{21} & z^{77} & z^{39} & z^{36} & z^{46} & z^{50} & z^{13} \\ z^{32} & z^{15} & z^{19} & z^{75} & z^{38} & z^{48} & z^{72} & z^{56} & z^{74} & z^{3} & z^{13} & z^{66} \end{pmatrix}$$

2.1.7.2 Additional parameter sets.

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

Table 2.2: 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 targeting 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	**(88, 80, 8, 10, 16)**	**1456 B**	**464 B**	**152 KB**	**153 KB**
	(85, 76, 9, 9, 16)	1726 B	407 B	135 KB	136 KB
	(80, 70, 10, 7, 16)	1941 B	304 B	110 KB	111 KB
	(77, 66, 11, 6, 16)	2194 B	255 B	96 KB	97 KB
	(86, 64, 13, 5, 16)	**2928 B**	**239 B**	**115 KB**	**117 KB**
	(98, 64, 16, 4, 16)	4368 B	220 B	148 KB	152 KB
	(102, 62, 21, 3, 16)	7177 B	177 B	153 KB	160 KB
	(110, 60, 30, 2, 16)	13966 B	134 B	167 KB	179 KB
	(146, 58, 58, 1, 16)*	49635 B	97 B	258 KB	304 KB
3	(123,114, 9, 13, 16)	2581 B	832 B	423 KB	425 KB
	(118,108, 10, 11, 16)	**2986 B**	**681 B**	**368 KB**	**371 KB**
	(115,104, 11, 10, 16)	3448 B	607 B	336 KB	339 KB
	(114,102, 12, 9, 16)	3994 B	545 B	324 KB	327 KB
	(113,100, 13, 8, 16)	4566 B	484 B	311 KB	315 KB
	(112, 98, 14, 7, 16)	5161 B	424 B	299 KB	303 KB
	(118, 96, 16, 6, 16)	6544 B	386 B	324 KB	330 KB
	(137, 96, 20, 5, 16)	10096 B	375 B	435 KB	444 KB
	(155, 96, 24, 4, 16)	14416 B	342 B	555 KB	567 KB
	(161, 94, 32, 3, 16)	24832 B	274 B	577 KB	599 KB
	(174, 92, 46, 2, 16)	49742 B	206 B	639 KB	684 KB
	(227, 90, 90, 1, 16)*	184291 B	146 B	964 KB	1138 KB
5	(153,142, 11, 13, 16)	4702 B	1035 B	814 KB	817 KB
	(154,142, 12, 12, 16)	**5554 B**	**964 B**	**823 KB**	**828 KB**
	(153,140, 13, 11, 16)	6386 B	882 B	801 KB	806 KB
	(152,138, 14, 10, 16)	7261 B	800 B	778 KB	784 KB
	(152,136, 16, 9, 16)	9264 B	724 B	765 KB	773 KB
	(151,134, 17, 8, 16)	10267 B	644 B	742 KB	751 KB
	(152,132, 19, 7, 16)	12556 B	572 B	739 KB	750 KB
	(168,132, 22, 6, 16)	16714 B	544 B	901 KB	915 KB
	(192,132, 27, 5, 16)	24964 B	520 B	1172 KB	1195 KB
	(212,130, 33, 4, 16)	36481 B	464 B	1401 KB	1434 KB
	(222,128, 43, 3, 16)	60560 B	373 B	1492 KB	1548 KB
	(241,126, 63, 2, 16)	127024 B	281 B	1676 KB	1795 KB
	(318,126,126, 1, 16)*	504079 B	199 B	2641 KB	3121 KB

* These parameter sets result in better key and signature sizes than the classic UOV signature scheme. This is possible because the $\mathcal{L}(\mathbf{\Lambda}, \mathbf{s})$ term means we are not constrained by claw-finding attacks, which would otherwise force $m \geq 64, 96$, or 128 for security levels 1, 3, and 5 respectively.

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, Intel AVX-512 optimized implementation written in C (C99) using the Galois Field New Instructions (GFNI), described in Section 3.4.
5. An additional, Arm NEON optimized implementation written in C (C99) and using assembly compiler intrinsics, as described in Section 3.6.
6. An additional, Arm Cortex-M4 optimized implementation written in C (C99) and assembly code, as described in Section 3.7.
7. An additional, simple textbook implementation written exclusively in Sage, described in Section 3.8.

The implementations 1-5 are delivered in a common code package, where each implementation can be compiled and built by providing the respective cmake options. For portability purposes, the code package does not make use of dynamic memory allocation or variable length arrays. The Cortex-M4 optimized implementation is delivered as a separate code package. Libraries supporting the NIST Signature API are built for each parameter set, along with a test harness to verify the Known Answer Tests (KAT) and applications to generate the KAT. For a detailed overview of the build options and built artifacts, we refer to the "README.md" file in the source code package submitted along with the specification.

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

3.1 Reference implementation

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

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

3.2 Optimized implementation

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

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

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

3.3 Intel AVX2 optimized implementation

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

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

3.4 Intel AVX-512/GFNI optimized implementation

The GFNI implementation requires the AVX-512F, AVX-512BW, and GFNI instruction set extensions, available on Intel server architectures since Ice Lake and on AMD architectures since Zen 4. It utilizes compiler assembly intrinsics for the AVX-512F and AVX-512BW instruction sets together with the Galois Field New Instructions (GFNI): the `vgf2p8affineqb` instruction applies an affine map to 64 bytes at once, which implements multiplication of a batch of $\mathbb{F}_{16}$ elements by a field element in a single instruction. Key expansion uses the vector AES extension (VAES) with 512-bit registers, processing 16 AES blocks per iteration.

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

3.5 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 GFNI implementation requires the AVX-512 and GFNI extensions and was therefore evaluated on Ice Lake only. 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`

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

Scheme	KeyGen	ExpandSK	ExpandPK	Sign	Verify	ExpandSK + Sign	ExpandPK + Verify
Reference Implementation							Generic portable C code, no AES-NI
$MAYO_1$	5,167,859	6,373,992	3,229,850	3,615,988	1,264,772	9,278,207	4,512,968
$MAYO_2$	4,808,997	5,893,250	2,433,909	1,440,137	632,553	4,847,643	3,074,526
$MAYO_3$	13,933,236	17,996,769	7,994,664	9,238,764	2,505,491	23,622,740	10,541,472
$MAYO_5$	33,650,822	44,269,421	17,860,703	20,617,052	5,303,196	53,648,143	23,240,866
Optimized Implementation							C code, using AES-NI (1st row), no AES-NI (2nd row)
$MAYO_1$	944,171	1,414,574	98,210	1,505,368	371,971	2,349,559	471,308
	4,115,066	4,538,498	3,229,555	1,585,133	452,403	5,597,050	3,711,115
$MAYO_2$	977,728	1,432,197	73,976	616,473	94,217	1,052,726	169,372
	3,309,175	3,788,996	2,433,636	678,604	158,122	3,484,220	2,613,840
$MAYO_3$	2,754,470	4,555,748	278,284	3,894,516	863,819	6,245,908	1,158,512
	10,366,967	12,102,577	7,852,145	4,024,437	996,462	13,991,516	8,941,771
$MAYO_5$	8,485,033	13,711,750	620,230	10,317,959	1,507,385	18,541,749	2,137,340
	25,412,198	30,697,826	17,551,275	10,562,231	1,763,493	35,822,699	19,531,982
AVX2 Optimized Implementation							AVX2 compiler intrinsics and using AES-NI
$MAYO_1$	190,996	246,989	98,163	238,663	142,685	466,134	242,382
$MAYO_2$	152,835	180,830	74,085	121,696	37,575	232,389	112,449
$MAYO_3$	451,465	634,454	237,946	463,447	285,733	1,015,330	528,461
$MAYO_5$	1,137,678	1,676,132	531,539	1,020,867	730,536	2,390,175	1,254,135

- Optimized implementation (without AES-NI): `-DMAYO_BUILD_TYPE=opt -DENABLE_AESNI=OFF`
- AVX2 implementation (using AES-NI): `-DMAYO_BUILD_TYPE=avx2 -DENABLE_AESNI=ON`
- GFNI implementation (using AES-NI): `-DMAYO_BUILD_TYPE=gfni -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 five configurations. We see that the use of AES-NI significantly speeds up the overall performance in operations using key expansion. On CPUs supporting the vector AES extension (VAES), such as Ice Lake, the AES-CTR expansion in the AES-NI-enabled builds processes 8 blocks per iteration using 256-bit VAES, and 16 blocks per iteration using 512-bit VAES in the GFNI implementation.

On Ice Lake, the AVX2 optimizations lead to a speed-up factor over the optimized implementation between approx. $10\times$-$22\times$ in KeyGen, $6\times$-$12\times$ in Signing (+ExpandSK) and $2\times$-$3.3\times$ in Verifying (+ExpandPK). The GFNI/AVX-512 implementation further improves over AVX2 by a factor of approx. $1.3\times$-$1.8\times$ in KeyGen, $1.4\times$-$2.0\times$ in Signing (+ExpandSK), and $1.2\times$-$1.8\times$ in Verifying (+ExpandPK).

The fastest results on Intel x86-64 are on the 2.0 GHz Ice Lake platform on which $MAYO_2$ performs KeyGen in 26.9 µs, Signing (+ExpandSK) in 50.6 µs, and Verifying (+ExpandPK) in 20.9 µs. Batch signing (without expandSK) takes 31.3 µs, and batch verification (without expandPK) takes 11.4 µs.

3.6 Arm NEON implementation

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

Table 3.2: MAYO performance in CPU cycles on an Intel i7-6700T CPU (**Skylake**) at 2.80GHz. The library was compiled on Debian with clang version 19.1.7. Results are the median of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	Sign	Verify	ExpandSK + Sign	ExpandPK + Verify
Reference Implementation							Generic portable C code, no AES-NI
$MAYO_1$	4,012,533	4,950,797	2,492,558	2,851,294	965,653	7,159,516	3,491,187
$MAYO_2$	3,726,147	4,583,953	1,875,917	1,106,671	495,116	3,687,332	2,390,638
$MAYO_3$	10,833,354	14,065,686	6,201,077	7,300,742	2,024,631	18,312,475	8,282,799
$MAYO_5$	26,316,370	34,582,953	13,849,483	16,378,889	4,348,471	41,779,855	18,387,502
Optimized Implementation							C code, using AES-NI (1st row), no AES-NI (2nd row)
$MAYO_1$	839,914	1,330,524	76,314	1,180,725	291,940	1,831,210	370,396
	3,257,589	3,745,745	2,489,524	1,242,112	350,116	4,311,714	2,853,263
$MAYO_2$	765,983	1,136,061	57,658	454,227	70,759	817,049	131,986
	2,590,963	2,968,391	1,888,783	501,577	119,449	2,690,367	2,000,812
$MAYO_3$	2,478,116	4,148,483	216,892	2,993,275	666,111	4,789,861	894,661
	8,385,023	10,020,376	6,074,495	3,100,651	780,356	10,771,213	6,871,991
$MAYO_5$	6,595,713	10,666,573	482,675	8,026,295	1,177,566	14,340,039	1,696,101
	19,797,715	23,962,159	13,705,979	8,218,205	1,370,990	27,741,059	15,017,727
AVX2 Optimized Implementation							AVX2 compiler intrinsics and using AES-NI
$MAYO_1$	149,119	194,658	76,312	185,144	112,043	363,653	188,988
$MAYO_2$	123,843	152,007	59,295	94,366	29,885	175,504	86,956
$MAYO_3$	361,791	489,873	184,919	363,708	220,422	789,086	407,824
$MAYO_5$	890,231	1,308,126	413,305	786,056	575,624	1,852,715	969,107

mentation is found at https://github.com/PQCMayo/MAYO-C. Table 3.4 shows benchmark results for M3.

3.7 Arm Cortex-M4 implementation

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

For benchmarking, we use the ST NUCLEO-L4R5ZI development board which features a STM32L4R5ZI Cortex-M4 CPU with 2 MB of flash memory and 640 KB of SRAM. We make use of the benchmarking framework PQM4 [KPR+] and use their implementations of Keccak and AES, i.e., we use the `Armv7-M` Keccak implementation from the XKCP [DHP+] and the T-table AES implementation of Stoffelen and Schwabe [SS16]. Note that AES is only used for expanding public values and, hence, using the T-table implementation (instead of the slower constant-time bit-sliced implementation) is acceptable even on Arm Cortex-M4 platforms with a data cache. All implementations were compiled with `-O3` using the Arm GNU toolchain (`arm-none-eabi-gcc`, version 13.2.1). Table 3.5 shows the benchmark results for the Cortex-M4.

3.8 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.

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

Scheme	KeyGen	ExpandSK	ExpandPK	Sign	Verify	ExpandSK + Sign	ExpandPK + Verify
Reference Implementation							Generic portable C code, no AES-NI
$MAYO_1$	5,172,820	6,201,866	3,223,078	3,306,404	1,053,614	8,763,750	4,350,776
$MAYO_2$	4,659,734	5,650,348	2,434,734	1,294,794	623,236	4,721,618	3,035,448
$MAYO_3$	13,610,732	17,298,180	8,035,466	8,283,834	2,257,850	22,405,426	10,220,548
$MAYO_5$	32,396,202	42,482,530	17,968,264	18,447,278	4,040,598	50,883,416	21,776,928
Optimized Implementation							C code, using AES-NI (1st row), no AES-NI (2nd row)
$MAYO_1$	890,892	980,834	25,014	1,818,942	302,308	2,531,962	320,660
	4,073,942	4,182,326	3,220,424	1,907,088	381,498	5,793,752	3,615,548
$MAYO_2$	1,599,408	989,372	19,016	494,984	73,298	909,718	98,106
	4,006,542	3,396,260	2,421,440	558,318	137,004	3,399,626	2,582,854
$MAYO_3$	3,424,220	3,275,580	92,268	4,717,688	694,864	6,945,436	802,626
	11,166,630	11,019,132	7,820,312	4,869,234	826,722	14,850,856	8,695,386
$MAYO_5$	6,943,014	11,741,570	206,726	9,091,036	1,378,662	15,877,932	1,558,240
	24,181,198	29,036,982	17,493,078	9,331,540	1,610,022	33,357,318	19,172,966
AVX2 Optimized Implementation							AVX2 compiler intrinsics and using AES-NI
$MAYO_1$	87,002	134,390	25,048	166,294	93,384	289,238	119,452
$MAYO_2$	72,552	95,440	19,020	90,578	29,174	140,176	49,064
$MAYO_3$	197,720	319,774	60,652	314,548	182,834	597,392	244,284
$MAYO_5$	571,546	1,030,476	135,318	721,430	503,884	1,515,402	639,410
GFNI/AVX-512 Optimized Implementation							AVX-512 and GFNI compiler intrinsics, AES-CTR using VAES
$MAYO_1$	66,210	77,112	24,548	97,776	49,056	166,390	73,926
$MAYO_2$	53,874	59,778	18,654	62,512	22,772	101,126	41,890
$MAYO_3$	149,334	178,216	59,582	175,296	92,266	337,246	152,066
$MAYO_5$	325,646	436,268	132,796	375,056	215,364	752,656	348,120

Table 3.4: MAYO performance of the NEON optimized implementation in CPU cycles on an Apple M3. The library was compiled with the Apple clang toolchain (`clang-1700.4.4.1`). Results are the average of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	Sign	Verify	ExpandSK + Sign	ExpandPK + Verify
$MAYO_1$	123,271	155,312	63,395	238,637	149,655	381,390	213,224
$MAYO_2$	112,322	141,107	47,918	132,948	80,035	214,151	128,003
$MAYO_3$	334,458	457,789	152,837	512,172	358,019	892,572	512,092
$MAYO_5$	795,090	1,083,099	341,523	954,674	692,657	1,789,597	1,033,879

Table 3.5: MAYO performance in CPU cycles on the **Arm Cortex-M4** (STM32L4R5ZI). The library was compiled with the Arm GNU toolchain (`arm-none-eabi-gcc` 13.2.1). Results are the average of 1000 benchmark runs.

Scheme	KeyGen	ExpandSK	ExpandPK	Sign	Verify	ExpandSK + Sign	ExpandPK + Verify
$MAYO_1$	9,631,297	9,786,056	6,911,431	8,322,944	4,627,416	19,223,231	11,539,985
$MAYO_2$	8,898,780	8,444,298	5,207,345	3,620,090	1,681,028	10,312,833	6,888,289
$MAYO_3$	27,169,341	29,677,229	18,077,967	17,535,468	10,397,669	43,708,718	28,475,402

Chapter 4

Known Answer Test values

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

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

Chapter 5

Security Analysis

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

5.1 Hard Problems underlying MAYO

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

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

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

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

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.

Definition 2 (Multi-Target Whipped MQ problem)**.** *For some matrices $\{\mathbf{E}_{ij}\}_{0\le i\le j<k} \in {\mathbb{F}_q}^{m\times m}$, let*

$$\mathcal{P}^\star(\mathbf{s}_0,\cdots,\mathbf{s}_{k-1}) := \sum_{i=0}^{k-1}\mathbf{E}_{ii}\mathcal{P}(\mathbf{s}_i) + \sum_{0\le i<j\le k-1}\mathbf{E}_{ij}\mathcal{P}'(\mathbf{s}_i,\mathbf{s}_j)\,, \text{ and}$$

$$\mathcal{L}(\mathbf{\Lambda},\mathbf{s}_0,\cdots,\mathbf{s}_{k-1}) := \sum_{i=0}^{k-1}\mathbf{E}_{i0}\mathbf{\Lambda}(\mathbf{s}_i)\,,$$

for some random $\mathcal{P} \in \mathsf{MQ}_{n,m,q}$. Given access to an unbounded number of random pairs $(\mathbf{t}_i, \mathbf{\Lambda}_i) \in \mathbb{F}_q^m \times \mathbb{F}_q^{m\times n}$ for $i \in \mathbb{N}$, the multi-target whipped MQ problem asks to compute $(I, \mathbf{s} = (\mathbf{s}_0,\ldots,\mathbf{s}_{k-1}))$, such that

$$\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}_I,\mathbf{s}) = \mathbf{t}_I\,.$$

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

$$\mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{A}) = \Pr\left[\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}_I, \mathbf{s}) = \mathbf{t}_I \;\middle|\; \begin{array}{c} \mathcal{P} \leftarrow \mathsf{MQ}_{n,m,q} \\ \{(\mathbf{t}_i, \mathbf{\Lambda}_i)\}_{i\in\mathbb{N}} \leftarrow \left(\mathbb{F}_q^m \times \mathbb{F}_q^{m\times n}\right)^{\mathbb{N}} \\ (I, \mathbf{s}) \leftarrow \mathcal{A}^{\mathbf{t}_i, \mathbf{\Lambda}_i}(\mathcal{P}) \end{array}\right].$$

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:*

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

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. *Let $\mathcal{L} : \mathbb{F}_q^{nk} \to \mathbb{F}_q^m$ be any fixed linear map. For $0 \leq i \leq j < k$, let the $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ be matrices such that*

$$\mathbf{E} = \begin{pmatrix} \mathbf{E}_{00} & \mathbf{E}_{01} & \dots & \mathbf{E}_{0,k-1} \\ \mathbf{E}_{01} & \mathbf{E}_{11} & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{E}_{0,k-1} & \dots & \dots & \mathbf{E}_{k-1,k-1} \end{pmatrix}$$

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 $\mathbb{F}_q^{n-o} \times \{0\}^o$ are chosen uniformly at random, then as a function of $\{\mathbf{o}_i\}_{i\in[k]} \in O$ the affine map

$$\mathcal{P}^\star(\mathbf{v} + \mathbf{o}) + \mathcal{L}(\mathbf{v} + \mathbf{o}) = \sum_{i=0}^{k-1} \mathbf{E}_{ii}\mathcal{P}(\mathbf{v}_i + \mathbf{o}_i) + \sum_{0\leq i<j\leq k-1} \mathbf{E}_{ij}\mathcal{P}'(\mathbf{v}_i + \mathbf{o}_i, \mathbf{v}_j + \mathbf{o}_j) + \mathcal{L}(\mathbf{v} + \mathbf{o})$$

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 $\mathbf{Z} \in (\mathbb{F}_q[z]/(f(z)))^{k\times k}$ be a matrix as in Section 2.1.1. We instantiated the matrices $\mathbf{E}_{ij} \in \mathbb{F}_q^{m\times m}$ for $i, j \in [k]$ as the matrix that corresponds to multiplication by $\mathbf{Z}_{ij}$ in $\mathbb{F}_q[z]/f(z)$. The requirement that $\mathbf{Z}$ is invertible implies that the matrix $\mathbf{E}$ is non-singular, so Lemma 1 indeed applies to our instantiation of MAYO.

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

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

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

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

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

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

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

1. Let Game_0 be $\mathcal{A}$'s EUF-CMA game against the MAYO signature scheme. By definition, we have that $\Pr[\mathsf{Game}_0() = 1] = \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{n,m,o,k,q}(\mathcal{A})$.

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 $\mathsf{M}\|\mathsf{R}\|\mathsf{seed}_{\mathsf{sk}}$ or $\mathsf{M}\|\mathsf{R}\|\mathsf{seed}'_{\mathsf{sk}}$, which happens with probability at most $2Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$ ($\mathsf{seed}_{\mathsf{sk}}$ has $8\mathsf{sk\_seed\_bytes}$ bits of min-entropy), then its view in Game_0 and Game_1 is identical. Therefore, we have $\Pr[\mathsf{Game}_1() = 1] \geq \Pr[\mathsf{Game}_0() = 1] - 2Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$.

3. Game_2 is the same as Game_1, but it simulates the random oracle SHAKE256 differently. Game_2 simulates the random oracle by maintaining a list L. When $\mathcal{A}$ makes a query on a message M, if there is an entry $(\mathsf{M}, \mathbf{t}, \mathbf{\Lambda}, \star)$ in L, the game answers with $(\mathbf{t}, \mathbf{\Lambda})$; otherwise, it forwards the query to the SHAKE256 oracle of the EUF-CMA game to receive $(\mathbf{t}, \mathbf{\Lambda})$, inserts $(\mathsf{M}, \mathbf{t}, \mathbf{\Lambda}, \perp)$ in L and answers with $(\mathbf{t}, \mathbf{\Lambda})$. When a signing query is made, Game_2 derives $\mathsf{M\_digest}$ and salt. It then:

   - Aborts if there is an entry $(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \mathbf{t}, \mathbf{\Lambda}, \perp)$ in L.
   - If there is an entry $(\mathsf{M\_digest}\|\mathsf{salt}, \mathbf{t}, \mathbf{\Lambda}, \mathbf{s})$, it answers the query with the signature $\mathsf{Encode}_{vec}(\mathbf{s})\|\mathsf{salt}$ (a signature derived from the found entry).
   - If there is no such entry in L, the game picks $\mathbf{t}$ and $\mathbf{\Lambda}$ uniformly at random, runs the signing algorithm for $\mathbf{t}$ to get a new $\mathbf{s}$, inserts $(\mathsf{M\_digest} \,\|\, \mathsf{salt}, \mathbf{t}, \mathbf{\Lambda}, \mathbf{s})$ in L, and outputs $\mathsf{Encode}_{vec}(\mathbf{s}) \,\|\, \mathsf{salt}$.

   Since there are at most $Q_h + Q_s$ entries of the form $(\mathsf{M}, \mathbf{t}, \mathbf{\Lambda}, \perp)$ in L and there are at most Q_s signing queries, the probability of an abort is at most $(Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}}$. 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 chooses a random $\mathbf{\Lambda}$, and then 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 $\mathcal{P}^\star(\mathbf{v} + \cdot) + \mathcal{L}(\mathbf{\Lambda}, \cdot) : O^k \to \mathbb{F}_q^m$ has full rank. Then, instead of picking $\mathbf{t}$ at random and sampling a random solution $\mathbf{o}$ to $\mathcal{P}^\star(\mathbf{v} + \mathbf{o}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{v} + \mathbf{o}) = \mathbf{t}$, Game_3 picks $\mathbf{o} \in O^k$ uniformly at random and sets $\mathbf{t} \leftarrow \mathcal{P}^\star(\mathbf{v} + \mathbf{o}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{v} + \mathbf{o})$. Since $\mathcal{P}^\star(\mathbf{v} + \cdot) + \mathcal{L}(\mathbf{\Lambda}, \cdot)$ is a full-rank affine map, it does not affect the distribution of the signatures. The only difference is that, in Game_2, $\mathbf{v}$ is determined

by the output of the SHAKE256 random oracle on input $\mathsf{M} \parallel \mathsf{salt} \parallel \mathsf{seed_{sk}} \parallel \mathsf{ctr}$, whereas in Game_3 they are chosen at random. If the adversary does not make a random oracle query of the form $\mathsf{M} \parallel \mathsf{salt} \parallel \mathsf{seed_{sk}} \parallel \mathsf{ctr}$ (which it can do with at most a probability $Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$, since $\mathsf{seed_{sk}}$ 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 $\mathcal{P}^\star(\mathbf{v} + \cdot) + \mathcal{L}(\mathbf{\Lambda}, \cdot)$ has full rank. Let $\mathsf{Good\_v}$ be the event that arises when, for all the $\mathbf{v}$ chosen during the execution of the EUF-KOA game, the map $\mathcal{P}^\star(\mathbf{v} + \cdot) + \mathcal{L}(\mathbf{\Lambda}, \cdot)$ happens to have full rank. Then, Game_5, conditioned on $\mathsf{Good\_v}$ happening, is identical to Game_4. Lemma 1 states that the probability that $\mathcal{P}^\star(\mathbf{v} + \cdot) + \mathcal{L}(\mathbf{\Lambda}, \cdot)$ does not have full rank for a single $\mathbf{v}$ is at most B, so, by the union bound, we have:

$$\begin{aligned}\Pr[\mathsf{Game}_5() = 1] &= \Pr[\mathsf{Game}_5() = 1 \mid \mathsf{Good\_v}] \Pr[\mathsf{Good\_v}] + \Pr[\mathsf{Game}_5() = 1 | \neg \mathsf{Good\_v}] \Pr[\neg \mathsf{Good\_v}] \\ &\geq \Pr[\mathsf{Game}_4() = 1] \Pr[\mathsf{Good\_v}] \\ &\geq \Pr[\mathsf{Game}_4() = 1](1 - Q_s \mathsf{B}) \,.\end{aligned}$$

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 $\mathsf{M\_digest} \parallel \mathsf{salt}$, where $\mathsf{M\_digest}$ 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_5 but not for EUF-KOA game if $\mathsf{SHAKE256}(\mathsf{M}) = \mathsf{SHAKE256}(\mathsf{M}')$, where M' was one of the messages queried 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_5, 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:

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

□

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 $\mathsf{P1\_bytestring}$, $\mathsf{P2\_bytestring}$, $\mathsf{P3\_bytestring}$ of $\{\mathbf{P}^{(1)}_i\}_{i\in[m]}$, $\{\mathbf{P}^{(2)}_i\}_{i\in[m]}$, and $\{\mathbf{P}^{(3)}_i\}_{i\in[m]}$, respectively. $\mathcal{B}$ then derives $\mathsf{seed}_{\mathsf{pk}}$ as in the normal key generation algorithm, and runs $\mathcal{A}$ on input $\mathsf{pk} = (\mathsf{seed}_{\mathsf{pk}} \,\|\, \mathsf{P3\_bytestring})$, while faithfully simulating a random oracle SHAKE256, and an oracle AES-128-CTR that outputs $\mathsf{P1\_bytestring} \,\|\, \mathsf{P2\_bytestring}$ 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 $\{\mathbf{t}_i, \mathbf{\Lambda}_i\}_{i\in\mathbb{N}}$, it does the same thing as Game_1, except that instead of simulating a SHAKE256 random oracle honestly, $\mathcal{B}'$ outputs $(\mathbf{t}_i, \mathbf{\Lambda}_i)$ in response to the i-th unique random oracle query, truncated or extended with random bits to achieve the requested output length. If $\mathcal{A}$ outputs a message-signature pair $(\mathsf{M}, (\mathsf{salt}, \mathbf{s}))$, then $\mathcal{B}'$ checks if the signature is valid (simulating a random oracle query in the process). If the signature is valid, then $\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt}$ is one of the random oracle queries, say the I-th unique random oracle query. Then, $\mathcal{B}'$ outputs $(I, \mathbf{s})$. If the signature is invalid, $\mathcal{B}'$ aborts. The view of $\mathcal{A}$ in this game is the same as the view of $\mathcal{A}$ in Game_1, since $\mathcal{B}'$ simulates the random oracle perfectly. Therefore, $\mathcal{A}$ outputs a valid message-signature pair with probability $\Pr[\mathsf{Game}_1() = 1]$. Therefore, we have $\mathsf{Adv}^{\mathsf{MTWMQ}}_{\{\mathbf{E}_{ij}\},n,m,k,q}(\mathcal{B}'^{\mathcal{A}}) = \Pr[\mathsf{Game}_1() = 1]$.

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

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

□

5.3 Discussion of the advantage loss in the security proof

The security reduction from the previous section loses advantage by three additive terms $(Q_s + Q_h + 2)^2 2^{-8\mathsf{digest\_bytes}}$, $3Q_h 2^{-8\mathsf{sk\_seed\_bytes}}$, and $(Q_h + Q_s)Q_s 2^{-8\mathsf{salt\_bytes}}$, 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 $\mathsf{salt\_bytes}$ 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 17 of MAYO.Sign. This factor stems from the fact that the rejection sampling functionality introduces a small amount of information-theoretic leakage.

If $Q_s\mathsf{B} < 1/2$, then the term only results in a loss in advantage by a constant factor, so the leakage provably does not hurt the security of MAYO by much. If $Q_s\mathsf{B} > 1$, the security proof no longer makes any guarantees, but there does not seem to be any attack that can take advantage of the information-theoretic leakage. The *Oil and Vinegar* signature scheme suffers from the same problem, but with a bigger leakage due to the larger restarting probability of approximately $1/q$. After decades of cryptanalysis, no attacks are known that can efficiently make use of this leakage. For MAYO_3 and MAYO_5, the bound B is approximately 2^{-12}, so the security proof gives a meaningful guarantee as long as the adversary sees fewer than 2^{11} signatures. For MAYO_1 and MAYO_2, B is larger, approximately 2^{-4} and 2^{-8} respectively, so the security proof only remains meaningful for adversaries that request very few signatures, and gives essentially no guarantee for realistic values of Q_s. However, as explained above, we expect the MAYO signature scheme to remain secure even if the adversary has access to an unbounded number of signatures, even though the security proof no longer gives a meaningful guarantee.

5.4 Analysis of known attacks

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

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

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

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

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

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

is non-positive.

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

Table 5.1: Bit-complexity lower bounds for the state-of-the-art attacks against our proposed parameter sets. The Kipnis-Shamir, reconciliation, and p^ℓ-truncated reconciliation attacks are key-recovery attacks, and the direct attack is a universal forgery attack. We have omitted the intersection and p^ℓ-truncated intersection attacks since for our parameters they are much less efficient than the reconciliation attacks.

Parameter set (n, m, o, k)	Kipnis-Shamir	Reconciliation (D, k)	p^ℓ-Reconcile (ℓ, D, k)	Direct attack
MAYO$_1$ $(88, 80, 8, 10)$	311	202 (20, 14)	241 (3, 41, 2)	150
MAYO$_2$ $(86, 64, 13, 5)$	263	203 (14, 15)	203 (2, 31, 2)	145
MAYO$_3$ $(118, 108, 10, 11)$	416	262 (23, 22)	313 (3,53,2)	222
MAYO$_5$ $(154, 142, 12, 12)$	546	334 (30, 27)	396 (3, 66, 2)	296

Attacks exploiting the loss in the security proof.

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

Guessing seed$_{\text{sk}}$**.** 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 $2^{8\text{sk\_seed\_bytes}-1}$ attempts. We set sk_seed_bytes $= 24$, 32, or 40 for the parameter sets targeting security levels 1, 3, and 5 respectively. The bit length of seed$_{\text{sk}}$ 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. Assume $n - m \geq o$, which is the case for the MAYO parameter sets. The attacker can use $n-m$ random affine constraints to eliminate $n - m$ variables in the system $\mathcal{P}(\mathbf{x}) = 0$, and with probability q^{-n+m+o}, the resulting system will have a solution in O. Therefore, finding a vector in O reduces to repeatedly solving a system of m inhomogeneous multivariate quadratic equations in m variables, roughly q^{n-m-o} times. 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 $q^{n-m-o} \cdot \mathsf{XL\_Cost}_{m,m,q}$.

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:

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

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

Rectangular Minrank attack Furue and Ikematsu showed that the rectangular minrank attack is applicable to MAYO [FI23]. The idea behind rectangular Minrank attacks [Beu21] is to look at linear maps of the form $L_{\mathbf{x}} : \mathbb{F}_{16}^n \to \mathbb{F}_{16}^m : \mathbf{y} \mapsto P'(\mathbf{x}, \mathbf{y})$. It follows from the fact that $\mathcal{P}$ vanishes on O, that for any $\mathbf{o} \in O$, the map $L_{\mathbf{o}}$ has rank at most $n - o$, because O is included in its kernel. Therefore, if $n - o < m$, then $L_{\mathbf{o}}$ has an unusually small rank, since a random linear map from $\mathbb{F}_{16}^n$ to $\mathbb{F}_{16}^m$ has rank close to m. A strategy to find vectors in O is therefore to look for vectors $\mathbf{x}$ such that $L_{\mathbf{x}}$ has low rank, which is an instance of the minrank problem. For more details we refer to [FI23]. We use parameters such that $n - o \geq m$, which completely prevents the attack, since $L_{\mathbf{o}}$ will have full rank for most $\mathbf{o} \in O$.

Wedge attack During the second round of the NIST project for additional signatures, Ran discovered a key recovery attack against UOV and its variants based on the wedge product in the exterior algebra [Ran26]. Initially the attack worked exclusively in characteristic two, but subsequently the attack was generalized (in multiple, non-equivalent ways) to odd characteristic. However, the attack remains more efficient in characteristic two. The attack can be interpreted as a variation on the XL algorithm. The attack requires the dimension of the secret oil space to be sufficiently large, which explains why the attack was only applicable to the round 2 version of MAYO_2, and not to the other parameter sets. A drawback of the attack is that it requires to do linear algebra on a single huge implicitly defined matrix, which is costly to parallelize, and requires a lot of memory and costly memory accesses. This makes the attack less practical than other key-recovery attacks, which usually have a guessing component which makes them trivial to parallelize, as different guesses can be processed independently by separate machines. In the third round submission, we have adapted the MAYO_2 parameter set such that the wedge attack is no longer applicable. The generalization of the wedge attack by Furue and Ikematsu [FI26] does apply, but for our chosen parameter sets its cost is comparable to, or higher than, the cost of the reconciliation attack (see Table 5.1).

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 $\mathsf{salt} \in \mathcal{B}^{\mathsf{salt\_bytes}}$ such that $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$, where $\mathbf{t}$ and $\mathbf{\Lambda}$ are determined by $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt})$. 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 $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$ with $\mathbf{t}, \mathbf{\Lambda}$ consistent with 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 $\mathsf{salt} \in \mathcal{B}^{\mathsf{salt\_bytes}}$, computes $\mathbf{t}$ and $\mathbf{\Lambda}$ from $\mathsf{SHAKE256}(\mathsf{SHAKE256}(M) \,\|\, \mathsf{salt})$ and solves for $\mathbf{s} \in \mathbb{F}_{16}^{n \times k}$ such that $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$. There are at present no known algorithms that can take advantage of the structure of the system $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$ to find a solution more efficiently. Therefore, to estimate the cost of this attack we will assume that $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$ behaves like a generic system of m quadratic equations in kn variables.

The system $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$ is underdetermined, i.e. the number of variables nk is larger than the number of equations m, and therefore the system is expected to have many solutions. The problem of finding a solution to an underdetermined system of equations clearly reduces to the problem of solving one of many determined systems: by fixing $nk - m$ variables in $\mathcal{P}^\star(\mathbf{s}) + \mathcal{L}(\mathbf{\Lambda}, \mathbf{s}) = \mathbf{t}$ one gets a smaller determined system, and any solution to the smaller system lifts to a solution of the original system. A line of work by Thomae, Wolf, Furue, Nakamura, Takagi, Hashimoto, May, Ostuzzi, Ressler, Asanuma, Chen, and Sakata [TW12, FNT21, Has21, MOR26, Ost26, ACF+26] has shown that if a system of quadratic equations is sufficiently underdetermined, then one can improve over this naive approach by doing a change of variables on $\mathbf{s}$, and then fixing variables more cleverly.

For sufficiently underdetermined systems, the most efficient algorithm seems to be the generalization of Hashimoto's algorithm by Asanuma, Chen, Furue, Sakata, and Takagi [ACF+26]. We have used their methodology to calculate concrete estimates of the cost of this attack in Table 5.1 even though we believe that their methodology slightly underestimates the cost of the attack, since it ignores the cost of making variable substitutions and eliminating variables, which is not always negligible.

Claw finding attacks. Earlier versions of MAYO did not have the $\mathcal{L}(\mathbf{\Lambda}, \mathbf{s})$ term in the verification equation. An attacker could then forge a signature for message M by finding a salt and $\mathbf{s}$ such that $\mathcal{P}^\star(\mathbf{s}) = \mathsf{SHAKE256}(\mathsf{SHAKE256}(M)||\mathsf{salt})$, which is a claw-finding problem: One can pick $\mathbf{s}$ and (M, salt) independently until both sides of the verification equation match. Naively, this gives an attack with complexity $\tilde{O}(q^{m/2})$, which can be improved to $\tilde{O}(q^{(m-1)/2})$ using the fact that $\mathcal{P}^\star$ is homogeneous[1].

In August 2026, three different research teams informed us, independently, of a more efficient claw-finding attack, using polar-isotropic spaces, all using AI cryptanalysis. These attacks reduce the cost of the claw-finding attack by up to a factor $2^{20}, 2^8, 2^{22}, 2^{24}$ for MAYO_1, MAYO_2, MAYO_3, and MAYO_5 respectively, which brings them slightly below the target security levels. In the third-round submission we have included the linear term $\mathcal{L}(\mathbf{\Lambda}, \mathbf{s})$ to the verification equation to avoid these collision attacks. Note that $\mathcal{L}$ is a universal hashing family, in the sense that for any non-zero $\mathbf{s}$ we have that $\mathcal{L}(\mathbf{\Lambda}, \mathbf{s})$ is uniformly random in $\mathbb{F}_q^m$, for uniformly random $\mathbf{\Lambda}$.

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,

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

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. The performance of our optimized implementations of MAYO is similar to that of lattice-based signatures on modern Intel x86-64 or ARM CPUs, and is only slower by a small factor on embedded platforms such as Cortex-M4 CPUs.

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

Wide security margin against known attacks. State-of-the-art attacks against MAYO are well-understood and easy to analyze. We pick parameters using a conservative methodology that only focuses on gate count and ignores the cost of memory accesses and parallelization.

Friendly to advanced cryptography. MAYO works over finite fields of characteristic 2 and only uses basic arithmetic operations, without any rejection sampling or other data-dependent branching. This makes MAYO well suited for use inside advanced cryptographic protocols, such as evaluating the signing algorithm in a multi-party computation protocol, thresholdizing the signature scheme, or proving knowledge of a signature in a zero-knowledge proof system.

Limitations

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

[ACF+26] Hideki Asanuma, Yilong Chen, Hiroki Furue, Kosuke Sakata, and Tsuyoshi Takagi. Improved complexity estimates for underdetermined MQ systems via generalized variable partitioning. Cryptology ePrint Archive, Paper 2026/1054, 2026.

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

[AFI+24] Rika Akiyama, Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Koha Kinjo, Haruhisa Kosuge, Satoshi Nakamura, Shingo Orihara, Tsuyoshi Takagi, and Kimihiro Yamakoshi. QR-UOV. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

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

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

[Beu22] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 355–376. Springer, Cham, September / October 2022.

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

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

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

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

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

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

[FI26] Hiroki Furue and Yasuhiko Ikematsu. Key recovery attacks on UOV using p^ℓ-truncated polynomial rings. Cryptology ePrint Archive, Paper 2026/298, 2026.

[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, *PQCrypto 2021*, pages 65–78. Springer, Cham, 2021.

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

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

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

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

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

[KX24] Haruhisa Kosuge and Keita Xagawa. Probabilistic hash-and-sign with retry in the quantum random oracle model. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part I*, volume 14601 of *LNCS*, pages 259–288. Springer, Cham, April 2024.

[MOR26] Alexander May, Massimo Ostuzzi, and Henrik Ressler. Just guess: Improved (quantum) algorithm for the underdetermined MQ problem. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part IV*, volume 16545 of *LNCS*, pages 334–362. Springer, Cham, May 2026.

[Ost26] Massimo Ostuzzi. Pseudo-oil subspaces and the geometry of underdetermined MQ problems. Cryptology ePrint Archive, Paper 2026/1122, 2026.

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

[Ran26] Lars Ran. Wedges, oil, and vinegar - an analysis of UOV in the exterior algebra. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part IV*, volume 16545 of *LNCS*, pages 363–388. Springer, Cham, May 2026.

[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, Cham, August 2016.

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

[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.