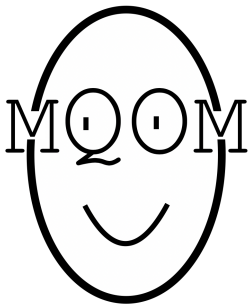
MQOM: MQ on my Mind

Algorithm Specifications and Supporting Documentation (Version 1.0)

Thibauld Feneuil Matthieu Rivain

May 31, 2023

CryptoExperts



Contents

1 Introduction

This specification presents the MQOM (MQ-On-my-Mind) digital signature scheme. This scheme has been designed by applying the "Multiparty Computation in the Head" (MPC-in-the-Head, or MPCitH) paradigm to the Multivariate Quadratic (MQ) problem. The MPC-in-the-Head paradigm was introduced in [IKO+07] and recently gained popularity, notably following the Picnic signature scheme proposal to the previous NIST call for post-quantum algorithms [**picnic**]. We design a specific MPC protocol to verify the solution of an MQ instance which is well suited for application of the MPCitH paradigm. The MQOM MPC protocol is obtained by improving previous techniques from [Fen22; BDK+21; DOT21]. We further use generic techniques to improve the MPCitH transform, namely seed trees [KKW18] and hypercube party computation [AGH+23].

Organization of the document. Section 2 gives a high-level description of MQOM MPC protocol and signature scheme. Section 3 provides a detailed description of the key generation, signature and verification algorithms and their underlying subroutines. In Section 4, we explain how we chose the parameters, we explicit our proposed instances and their achieved performances. Section 5 provides a security analysis of the MQOM signature scheme. Finally, we conclude by listing some advantages and limitations of MQOM in Section 6.

2 High-level description of the MQOM signature scheme

2.1 Notations

Table 1: Notations and parameters of the MQOM scheme.

MQ parameters:	
q	Size of the base field $\mathbb{F}_q$.
n	Number of unknowns.
m	Number of equations.
MPC & signature parameters:	
λ	Security parameter.
N	Number of parties (number of shares).
n_1	Number of coordinates per chunks of x and w.
n_2	Number of chunks in x and w.
η	Extension degree for the field $\mathbb{F}_{q^\eta}$ used in the MPC protocol.
τ	Number of repetitions.
MQ instance:	
$\{A_i, b_i\}$	Equations of the MQ instance.
x	Secret MQ solution (in $\mathbb{F}_q^n$).
y	Output of the MQ instance (in $\mathbb{F}_q^m$): $y_i = x^\intercal A_i x + b_i^\intercal x \quad \forall i \in [1:m]$.
MPC protocol:	
$[\![\cdot]\!] = ([\![\cdot]\!]_1, \ldots, [\![\cdot]\!]_N)$	Sharing.
$\gamma_1, \ldots, \gamma_m$	Random coefficients of $\mathbb{F}_{q^\eta}$ (first challenge).
r	Random evaluation point of $\mathbb{F}_{q^\eta} \setminus \{f_1, \ldots, f_{n_1}\}$ (second challenge).
w	Vector defined as $w := \left(\sum_{i=1}^m \gamma_i A_i\right) x$ (in $\mathbb{F}_{q^\eta}^n$).
z	Value defined as $z := \sum_{i=1}^m \gamma_i (y_i - b_i^\intercal x)$ (in $\mathbb{F}_{q^\eta}$).
$f_1, \ldots, f_{n_1}$	Elements of $\mathbb{F}_{q^\eta}$ for interpolation of the polynomials.
$X = (X_1, \ldots, X_{n_2})$	Polynomials interpolated from the chunks of x.
$W = (W_1, \ldots, W_{n_2})$	Polynomials interpolated from the chunks of w.
$a = (a_1, \ldots, a_{n_2})$	Random coefficients from the hint (in $\mathbb{F}_{q^\eta}$).
V	Vanishing polynomial: $V(u) = \prod_{i=1}^{n_1} (u - f_i)$.
$\tilde{W} = (W_1, \ldots, W_{n_2})$	Masked polynomials: $\tilde{W}_j(u) = W_j(u) + a_j \cdot V(u) \quad \forall j \in [1:n_2]$
$\alpha = (\alpha_1, \ldots, \alpha_{n_2})$	Evaluations of $\tilde{W}$ in r: $\alpha_j = \tilde{W}_j(r) \quad \forall j \in [1:n_2]$
$Q(u)$	Hint polynomial: $Q(u) = \sum_{j=1}^{n_2} X_j(u) \cdot \tilde{W}_j(u)$
$Q'(u)$	Truncated hint polynomial: $Q(u) = u \cdot Q'(u) + q_0$
$p = p_1 + (1 - p_1) \cdot p_2$	False positive probability.
MPCitH & signature:	
D	The dimension of the hypercube ($N = 2^D$).
$I_{(d,b)}$	Set of parties on a hypercube face.
i^*	Index of the non-opened party.
h_1, h_2, h_3	Fiat-Shamir hashes.
$\mathsf{seed_{eq}}$	Seed for the generation of the MQ equations $\{A_i, b_i\}$ (in $\{0,1\}^\lambda$).
salt	Salt used as auxiliary input of XOF, Hash and Commit (in $\{0,1\}^{2\lambda}$).
mseed	Master seed of the signature (in $\{0,1\}^\lambda$).
$\mathsf{rseed}^{[e]}$	Root seed for execution e (in $\{0,1\}^\lambda$).
$\mathsf{seed}_i^{[e]}$	Leaf seed for party i in execution e (in $\{0,1\}^\lambda$).
$\mathsf{com}_i^{[e]}$	Commitment of party i in execution e (in $\{0,1\}^{2\lambda}$).

2.2 The Multivariate Quadratic (MQ) problem

We recall the definition of the MQ problem (in matrix form) which is the core hardness assumption of the MQOM signature scheme.

Definition 1 (Multivariate Quadratic Problem)**.** *Let q be a prime (or a prime power) and let m, n be positive integers. The multivariate quadratic problem with parameters (q, m, n) is the following problem:*

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

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

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

2.3 The MQOM MPC protocol

In this section, we describe the MQOM MPC protocol which is at the core of the MQOM signature scheme. The MQOM MPC protocol runs a multi-party computation which verifies the correctness of a solution x to a public MQ instance $\big(\{A_i\}, \{b_i\}, y\big)$. The secret solution x is shared between N parties which, after running the protocol, either output Accept if the input sharing is believed to encode a correct MQ solution or Reject otherwise.

2.3.1 MPC model

The MQOM protocol relies on a standard model of the MPCitH paradigm as formalized *e.g.* in [FR22, Section 3.1]. In this model, the parties receive as input a sharing

$$[\![x]\!] = ([\![x]\!]_1, \ldots, [\![x]\!]_N)$$

of the secret witness x. Then, they perform a sequence of the following actions:

1. Invoke a randomness oracle $\mathcal{O}_R$. This oracle sends a random value γ to all the parties. In MPCitH context, these random values are provided by the verifier as challenges.

2. Invoke a hint oracle $\mathcal{O}_H$. For some function ψ, the hint is sampled as $\beta \leftarrow \psi(x, \gamma^1, \gamma^2, \ldots; r_\psi)$ where $\gamma^1, \gamma^2, \ldots$ are the previous outputs of $\mathcal{O}_R$ and where r_ψ is some fresh randomness. The hint oracle sends a random sharing $[\![\beta]\!]$ of the hint to the parties. In the MPCitH context, the hint is computed by the prover and the obtained shares are committed with the shares of the witness.

3. Compute and broadcast shares. The parties locally compute $[\![\alpha]\!] := [\![\varphi(v)]\!]$ from a sharing $[\![v]\!]$ for some is an $\mathbb{F}_q$-linear function φ. Then they broadcast the shares $[\![\alpha]\!]_1, \ldots, [\![\alpha]\!]_N$ and publicly reconstruct $\alpha = \varphi(v)$. This local computation process is denoted $[\![\varphi(v)]\!] = \varphi([\![v]\!])$. The function φ can depend on the previous random values from $\mathcal{O}_R$ and on the previous broadcasted values from $\mathcal{O}_H$.

After a given number of rounds, the parties broadcast a final sharing $[\![v]\!]$ and publicly recompute v. If $v = 0$, they output Accept, if $v \neq 0$, they output Reject.

The protocol is said perfectly complete, if on input a sharing $[\![x]\!]$ of the right MQ solution, the parties always output Accept. On the other hand, the protocol has *false positive probability* p, if the probability that the parties output Accept on input $[\![x]\!]$ corresponding to an incorrect MQ solution is at most p. The MQOM protocol is perfectly complete and has false positive probability which we exhibit in Section 2.3.4.

2.3.2 Principle of the MPC protocol

The parties aim to verify that their input sharing $[\![x]\!]$ corresponds to a solution $x \in \mathbb{F}_q^n$ of the following system:

$$\begin{cases} y_1 &= x^\intercal A_1 x + b_1^\intercal x \\ &\vdots \\ y_m &= x^\intercal A_m x + b_m^\intercal x \end{cases}$$

for a given MQ instance $\big(\{A_i\}, \{b_i\}, y\big)$.

As a first step of the MQOM protocol, we batch the equations of the MQ system as suggested in [Fen22]. Instead of checking the m equations separately, the parties verify a linear combination of these equations with coefficients $\gamma_1, \ldots, \gamma_m$ that are uniformly sampled by the randomness oracle $\mathcal{O}_R$ from a field extension $\mathbb{F}_{q^\eta}$. The MPC protocol shall then check that

$$\sum_{i=1}^{m} \gamma_i (y_i - x^\intercal A_i x - b_i^\intercal x) = 0. \tag{1}$$

If one of the equations of the MQ system is not satisfied, then Equation 1 is only satisfied with a probability $1/q^\eta$. Then, the above equality rewrites as

$$\begin{aligned} \sum_{i=1}^{m} \gamma_i (y_i - b_i^\intercal x) &= \sum_{i=1}^{m} \gamma_i (x^\intercal A_i x) \\ &= x^\intercal \Big(\sum_{i=1}^{m} \gamma_i A_i \Big) x \\ &= \langle x, w \rangle \quad \text{where} \quad w := \Big(\sum_{i=1}^{m} \gamma_i A_i \Big) x \end{aligned}$$

By defining $z := \sum_{i=1}^{m} \gamma_i (y_i - b_i^\intercal x)$ and $w := \big(\sum_{i=1}^{m} \gamma_i A_i\big)\, x$, checking Equation 1 is equivalent to checking $z = \langle x, w \rangle$. On receiving the random coefficients $\gamma_1, \ldots, \gamma_m$ the parties can locally compute a sharing $[\![w]\!]$ of w from the sharing $[\![x]\!]$.

As a second step of the MQOM protocol, the parties verify the inner product $z = \langle x, w \rangle$ for some public z from the sharings and $[\![x]\!]$ and $[\![w]\!]$. For this purpose, we introduce an inner-product verification protocol, using polynomial interpolation techniques as Banquet [BDK+21] or Limbo [DOT21] with a slightly improved communication.

The principle is to split the vectors x and z into n_2 chunks of n_1 coordinates for integers n_1, n_2 such that $n_1 \cdot n_2 \geq n$. The chunks are used to interpolate polynomials $X_1(u), \ldots, X_{n_2}(u) \in \mathbb{F}_q[u]$

and $W_1(u), \ldots, W_{n_2}(u) \in \mathbb{F}_{q^\eta}[u]$, which satisfy

$$\begin{cases} X_j(f_1) &= x_{(j-1)n_1+1} \\ &\vdots \\ X_j(f_{n_1}) &= x_{(j-1)n_1+n_1} \end{cases} \quad \text{and} \quad \begin{cases} W_j(f_1) &= w_{(j-1)n_1+1} \\ &\vdots \\ W_j(f_{n_1}) &= w_{(j-1)n_1+n_1} \end{cases} \tag{2}$$

for every $j \in [1 : n_2]$ where $f_1, \ldots, f_{n_1}$ are n_1 fixed distinct elements from $\mathbb{F}_q$. Let us stress that the X_j's are polynomials of degree $\deg(X_j) \leq n_1 - 1$ from $\mathbb{F}_q[u]$ while the W_j's are polynomials of degree $\deg(W_j) \leq n_1 - 1$ from $\mathbb{F}_{q^\eta}[u]$. By definition of those polynomials, we have the following equivalence

$$z = \langle x, w \rangle \quad \iff \quad z = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} X_j(f_i) \cdot W_j(f_i)$$

Defining the polynomial $Q_0(u) \in \mathbb{F}_{q^\eta}[u]$ as

$$Q_0(u) := \sum_{j=1}^{n_2} X_j(u) \cdot W_j(u) \tag{3}$$

we now have $z = \langle x, w \rangle$ if and only if $z = \sum_{i=1}^{n_1} Q_0(f_i)$.

In a nutshell, the MQOM MPC protocol works as follows:

- The parties locally computes sharings $[\![X_j]\!]$ and $[\![W_j]\!]$ by interpolation from $[\![x]\!]$ and $[\![w]\!]$ (which is possible since such interpolation is $\mathbb{F}_q$-linear);
- The parties invoke the hint oracle to obtain a sharing $[\![Q_0]\!]$ of the polynomial Q_0;
- At this stage, the parties aim to check that the two following holds:
  (1) the polynomial Q_0 from the hint oracle well satisfies Equation 3,
  (2) the polynomial Q_0 from the hint oracle well satisfies $z = \sum_{i=1}^{n_1} Q_0(f_i)$.
- To verify relation (1), the protocol relies on the Schwartz–Zippel lemma. Namely, Equation 3 is verified on a random evaluation point $r \in \mathbb{F}_{q^\eta} \setminus \{f_1, \ldots, f_{n_1}\}$. The parties proceed as follows:
  1. they request a random evaluation point $r \in \mathbb{F}_{q^\eta} \setminus \{f_1, \ldots, f_{n_1}\}$ from the randomness oracle $\mathcal{O}_R$;
  2. they locally compute $[\![\alpha_j]\!] = [\![W_j]\!](r)$ for all $j \in [1 : n_2]$;
  3. they broadcast $[\![\alpha_j]\!]$ and publicly recompute α_j for all $j \in [1 : n_2]$;
  4. they locally compute $[\![v_1]\!] = [\![Q_0]\!](r) - \sum_{j=1}^{n_2} \alpha_j \cdot [\![X_j]\!](r)$;
  5. they broadcast $[\![v_1]\!]$ and publicly recompute $v_1 = Q_0(r) - \sum_{j=1}^{n_2} \alpha_j \cdot X_j(r)$;
  6. they verify that $v_1 = 0$.
- To verify relation (2), the parties proceed as follows:
  1. they locally compute $[\![v_2]\!] = [\![z]\!] - \sum_{i=1}^{n_1} [\![Q_0]\!](f_i)$;
  2. they broadcast $[\![v_2]\!]$ and publicly recompute $v_2 = z - \sum_{i=1}^{n_1} Q_0(f_i)$;
  3. they verify that $v_2 = 0$.

Making the protocol private. The careful reader might have noticed a caveat in the above protocol: the broadcast shares leak information on the secret witness. Specifically, the publicly recomputed values α_j are evaluations of the polynomials $W_j(u)$ in a public point r which provide linear combinations of the vector w, namely linear combinations of the secret x. In order to ensure the zero-knowledge property when applying the MPCitH transformation, we should make the protocol $(N-1)$-private. Namely, one should not learn any information about x from any $N-1$ parties' views, including all the broadcast shares.

To deal with this issue we use a standard solution which consists in masking the polynomials W_j's as

$$\tilde{W}_j(u) := W_j(u) + a_j \cdot V(u)$$

where the a_j's are random elements of $\mathbb{F}_{q^\eta}$ obtained from the hint oracle $\mathcal{O}_H$ and where $V(u)$ is the *vanishing polynomial* of the interpolation set $\{f_1, \ldots, f_{n_1}\}$, which is defined as

$$V(u) := (u - f_1)(u - f_2) \cdots (u - f_{n_1}) \ .$$

The polynomial Q_0 is then replaced by the polynomial Q defined as

$$Q(u) := \sum_{j=1}^{n_2} X_j(u) \cdot \tilde{W}_j(u) \tag{4}$$

which –by definition of the vanishing polynomial– still satisfies the relation

$$z = \langle x, w \rangle \quad \iff \quad z = \sum_{i=1}^{n_1} Q(f_i) \ . \tag{5}$$

The protocol works the same way as before with Q in place of Q_0 but now the revealed evaluations $\tilde{W}_1(r)$, $\ldots$, $\tilde{W}_{n_1}(r)$ do not leak any information about the secret witness. Indeed, $\tilde{W}_j(r) = W_j(r) + a_j \cdot V(r)$ with $V(r) \neq 0$ since $r \notin \{f_1, \ldots, f_{n_1}\}$ and a_j uniformly random over $\mathbb{F}_{q^\eta}$.

Dealing with incomplete last chunk. In practice, the optimal parameters n_1, n_2 in terms of communication of the MPCitH-transformed protocol might be such that $n_1 \cdot n_2$ is strictly greater than n. In that case, the last of the n_2 chunks of x and w are of size smaller than n_1. Let $n_1' < n_1$, the size of the last chunks, such that $n = (n_2 - 1)n_1 + n_1'$. The last chunk polynomials X_{n_2} and W_{n_2} are defined as the polynomials of degrees $\deg(X_{n_2}) \leq n_1$ and $\deg(W_{n_2}) \leq n_1'$ respectively satisfying:

$$\begin{cases} X_{n_2}(f_1) = x_{(n_2-1)n_1+1} \\ \quad \vdots \\ X_{n_2}(f_{n_1'}) = x_{(n_2-1)n_1+n_1'} \\ X_{n_2}(f_{n_1'+1}) = 0 \\ \quad \vdots \\ X_{n_2}(f_{n_1}) = 0 \end{cases} \qquad \text{and} \qquad \begin{cases} W_{n_2}(f_1) = w_{(n_2-1)n_1+1} \\ \quad \vdots \\ W_{n_2}(f_{n_1'}) = w_{(n_2-1)n_1+n_1'} \end{cases} \tag{6}$$

This way, whatever the values taken by $W_{n_2}(f_i)$, we always have $X_{n_2}(f_i) \cdot \tilde{W}_{n_2}(f_i) = 0$ for $i \in [n_1' + 1 : n_1]$ so that Equation 5 still holds true.

Optimizing the communication. The MPC protocol as depicted above consists in checking that the publicly reconstructed values v_1 and v_2 satisfy $v_1 = v_2 = 0$, which is

$$\begin{cases} Q(r) - \sum_{j=1}^{n_2} \alpha_j \cdot X_j(r) = 0 \\ z - \sum_{i=1}^{n_1} Q(f_i) = 0 \end{cases} \tag{7}$$

By denoting q_0 the constant term of $Q(u)$ and letting $Q'(u)$ the degree-$(n_1 - 2)$ polynomial such that

$$Q(u) = u \cdot Q'(u) + q_0 \ , \tag{8}$$

the above system is equivalent to (swapping the two equations):

$$\begin{cases} q_0 = (n_1)^{-1} \Big(z - \sum_{i=1}^{n_1} f_i \cdot Q'(f_i) \Big) \\ v := r \cdot Q'(r) + q_0 - \sum_{j=1}^{n_2} \alpha_j \cdot X_j(r) = 0 \end{cases} \tag{9}$$

(assuming that n_1 is co-prime with q, the base field characteristic, which shall always be the case in our context). So rather than requesting a hint $[\![Q]\!]$ and broadcasting the shares of v_1 and v_2, the parties can instead request a hint $[\![Q']\!]$ and check the two equations simultaneously by

- locally computing the sharing $[\![q_0]\!]$ as

$$[\![q_0]\!] = (n_1)^{-1} \Big(z - \sum_{i=1}^{n_1} f_i \cdot [\![Q']\!](f_i) \Big)$$

- locally computing the sharing $[\![v]\!]$ as

$$[\![v]\!] = r \cdot [\![Q']\!](r) + [\![q_0]\!] - \sum_{j=1}^{n_2} \alpha_j \cdot [\![X_j]\!](r)$$

- broadcasting $[\![v]\!]$, publicly recomputing v and checking $v = 0$.

This way, the hint (Q' vs. Q) is shorter by one $\mathbb{F}_{q^\eta}$ element (which accounts in the ZK protocol's communication) and the broadcast is reduced by replacing $([\![v_1]\!], [\![v_2]\!])$ by $[\![v]\!]$ (which also accounts as one $\mathbb{F}_{q^\eta}$ element in the ZK protocol).

2.3.3 Description of the MPC protocol

Wrapping up the different tweaks described above, we obtain the MQOM MPC protocol which is formally depicted in Figure 1.

INPUT: The parties receive a sharing $[\![x]\!]$ of a solution x to an MQ instance $\big(\{A_i\}, \{b_i\}, y\big)$.

1. The parties invoke the randomness oracle $\mathcal{O}_R$ to get random $\gamma_1, \ldots, \gamma_m \in \mathbb{F}_{q^\eta}$.
2. The parties locally compute $[\![z]\!] = \sum_{i=1}^{m} \gamma_i (y_i - b_i^\intercal [\![x]\!])$.
3. The parties locally compute $[\![w]\!] = (\sum_{i=1}^{m} \gamma_i A_i)\, [\![x]\!]$.
4. The parties locally interpolate the chunk polynomials $[\![X_j]\!]$, $[\![W_j]\!]$, for $j \in [1 : n_2]$, as defined in Equation 2 and Equation 6 (for the last chunk).
5. The parties locally compute $[\![\tilde{W}_j]\!](u) = [\![W_j]\!](u) + [\![a_j]\!] \cdot V(u)$ where $V(u)$ is the vanishing polynomial over $\{f_1, \ldots, f_{n_1}\}$ defined as $V(u) := \prod_{i=1}^{n_1} (u - f_i)$.
6. The parties invoke the hint oracle $\mathcal{O}_H$ to get sharings $[\![a_1]\!]$, $\ldots$, $[\![a_{n_1}]\!]$ and $[\![Q']\!]$, for random $a_1, \ldots, a_{n_1} \in \mathbb{F}_{q^\eta}$ and Q' satisfying

$$u \cdot Q'(u) + q_0 := \sum_{j=1}^{n_2} \tilde{W}_j(u) \cdot X_j(u)$$

   for some $q_0 \in \mathbb{F}_{q^\eta}$ and with $\tilde{W}_j(u) = W_j(u) + a_j \cdot V(u)$ for every $j \in [1 : n_2]$.
7. The parties locally compute $[\![q_0]\!] = (n_1)^{-1} \Big(z - \sum_{i=1}^{n_1} f_i \cdot [\![Q']\!](f_i) \Big)$.
8. The parties invoke the randomness oracle $\mathcal{O}_R$ to get a random $r \in \mathbb{F}_{q^\eta} \backslash \{0, \ldots, n_1 - 1\}$.
9. The parties locally compute $[\![\alpha_j]\!] = [\![\tilde{W}_j]\!](r)$ for all $j \in [1 : n_2]$.
10. The parties broadcast $[\![\alpha_j]\!]$ and publicly recompute $\alpha_j \in \mathbb{F}_{q^\eta}$ for all $j \in [1 : n_2]$.
11. The parties locally compute $[\![v]\!] = r \cdot [\![Q']\!](r) + [\![q_0]\!] - \sum_{j=1}^{n_2} \alpha_j \cdot [\![X_j]\!](r)$.
12. The parties broadcast $[\![v]\!]$ and publicly recompute v.
13. The parties outputs ACCEPT if $v = 0$ and REJECT otherwise.

Figure 1: The MQOM MPC protocol.

2.3.4 False positive probability

Theorem 2.1. *If x is the right solution to the MQ instance $\big(\{A_i\},\{b_i\},y\big)$ and if $[\![Q']\!]$ is a genuinely generated as a sharing of Q' by the hint oracle $\mathcal{O}_H$, then the protocol always outputs* Accept. *If x does not satisfy the MQ instance, then whatever the hint returned by $\mathcal{O}_H$, the protocol outputs* Accept *with probability at most $p_1 + (1-p_1)\cdot p_2$ with*

$$p_1 := \frac{1}{q^\eta} \quad \text{and} \quad p_2 := \frac{2n_1 - 1}{q^\eta - n_1}.$$

Proof. The completeness of the protocol holds following the presentation of Section 2.3.2. Now, let us assume that x is not a solution of the MQ instance $\big(\{A_i\},\{b_i\},y\big)$. There are two cases:

1. Either $\langle w, x\rangle = z$, this case occurs with probability $p_1 = \frac{1}{q^\eta}$ over the randomness of $(\gamma_j)_j$;
2. Or, $\langle w, x\rangle \neq z$. Let us define the polynomial P of degree $\deg(P) \leq 2n_1 - 1$ defined w.r.t. Q', X, W and z as:

$$P(u) := u \cdot Q'(u) + q_0 - \sum_{j=1}^{n_2} W_j(u) \cdot X_j(u).$$

with $q_0 := (n_1)^{-1}\Big(z - \sum_{i=1}^{n_1} f_i \cdot Q'(f_i)\Big)$.

We have

$$\begin{aligned}
\sum_{i=1}^{n_1} P(f_i) &= \sum_{i=1}^{n_1} f_i \cdot Q'(f_i) + (n_1 \cdot q_0) - \sum_{j=1}^{n_2}\sum_{i=1}^{n_1} W_j(f_i)\cdot X_j(f_i) \\
&= \sum_{i=1}^{n_1} f_i \cdot Q'(f_i) + \left(z - \sum_{i=1}^{n_1} f_i \cdot Q'(f_i)\right) - \sum_{j=1}^{n_2}\sum_{i=1}^{n_1} W_j(f_i)\cdot X_j(f_i) \\
&\qquad\qquad \text{(by definition of } q_0\text{)} \\
&= \sum_{i=1}^{n_1} f_i \cdot Q'(f_i) + \left(z - \sum_{i=1}^{n_1} f_i \cdot Q'(f_i)\right) - \langle w, x\rangle \\
&\qquad\qquad \text{(by definition of } X \text{ and } W\text{)} \\
&= z - \langle w, x\rangle.
\end{aligned}$$

Since $\langle w, x\rangle \neq z$, we have that $\sum_{i=1}^{n_1} P(f_i) \neq 0$ which implies that P is not the null polynomial. Then, according to the Schwartz-Zippel Lemma, the probability that $v := P(r)$ is zero is at most $p_2 = \frac{2n_1-1}{q^\eta - n_1}$.

To sum up, we get

$$\begin{aligned}
\Pr[\text{Accept}] &\leq \Pr[\langle w, x\rangle = z] + \Pr[\langle w, x\rangle \neq z] \cdot \Pr[\text{Accept} \mid \langle w, x\rangle \neq z] \\
&= p_1 + (1 - p_1)\cdot p_2
\end{aligned}$$

□

2.4 The MQOM signature scheme

2.4.1 From MPC to signature

To obtain the MQOM signature scheme, we first apply the MPCitH transform to the MQOM MPC protocol. We use seed trees to optimize the communication [KKW18] and the hypercube technique to optimize the computation [AGH+23]. This gives us a zero-knowledge (ZK) proof of knowledge (PoK) protocol with soundness error about $1/N$ for N being the number of parties (which is slightly degraded by the false positive probability p of the MPC protocol). To make the soundness error negligible, we rely on parallel repetitions of this protocol. Finally, to turn the obtained sound ZK-PoK protocol into a signature scheme, we use the Fiat-Shamir transform. Those different steps are summarized hereafter.

The MPCitH transform. The MPC-in-the-Head paradigm turns the MQOM MPC protocol into a ZK-PoK with the following blueprint:

1. *Sharing and commitments:* the prover generates a sharing of the witness $[\![x]\!]$ and separately commits to each share;
2. *First challenge:* the verifier challenges the prover with the random coefficients $\gamma_1, \ldots, \gamma_m$ of the MPC protocol (in place of the randomness oracle $\mathcal{O}_R$);
3. *Hint and commitments:* the prover computes the hint $(a_1, \ldots, a_{n_2}, Q')$ as described in Section 2.3, generates a sharing of the hint $([\![a_1]\!], \ldots, [\![a_{n_2}]\!], [\![Q']\!])$ and commits to each share separately (*i.e.* the i-th shares $[\![\cdot]\!]_i$ of the different elements are committed all together while the shares of different indices are committed separately);
4. *Second challenge:* the verifier challenges the prover with the random verification point r of the MPC protocol (in place of the randomness oracle $\mathcal{O}_R$);
5. *MPC simulation:* the prover runs the MPC protocol *in their head* to compute the shares broadcasted by the parties, $[\![\alpha_1]\!]$, ..., $[\![\alpha_{n_2}]\!]$, $[\![v]\!]$, and send them to the verifier;
6. *Third challenge:* the verifier challenges the prover to open the views of the parties of indices $[1 : N] \setminus \{i^*\}$;
7. *Views opening:* the prover returns the shares $[\![x]\!]_i$, $[\![a_1]\!]_i$, ..., $[\![a_{n_2}]\!]_i$, $[\![Q']\!]_i$ for every $i \in [1 : N] \setminus \{i^*\}$;

   NB: The above shares are called the input shares (witness share and hint shares) of the party i. From the input shares of the party i, and given the previously received broadcast shares, the verifier can now fully recompute the view of party i, for every $i \in [1 : N] \setminus \{i^*\}$.
8. *Verification:* the verifier checks the consistency of the commitments and the MPC computation for the revealed parties. Namely they:
   - verify the commitments of the opened witness shares,
   - verify the commitments of the opened hint shares,
   - recompute the broadcast shares for each party $i \in [1 : N] \setminus \{i^*\}$ and verify that they match the received broadcast shares from the prover,
   - verify that the plain broadcast value v equals 0.

MPCitH optimization: Seed tree. As suggested in [KKW18], we use a seed tree (a.k.a. a puncturable PRF) to generate and commit the shares. The principle is to derive the input shares (witness and hint shares) of each party i pseudorandomly from a random seed seed_i, with the following pattern:

$$\begin{array}{lllll} \mathsf{seed}_1 & \longrightarrow & [\![x]\!]_1, & [\![a]\!]_1, & [\![Q']\!]_1 \\ \vdots & & \vdots & \vdots & \vdots \\ \mathsf{seed}_{N-1} & \longrightarrow & [\![x]\!]_{N-1}, & [\![a]\!]_{N-1}, & [\![Q']\!]_{N-1} \\ \mathsf{seed}_N & \longrightarrow & & [\![a]\!]_N & \end{array}$$

where $[\![a]\!]_i = ([\![a_1]\!]_i, \ldots, [\![a_{n_2}]\!]_i)$.

For the last party, only the share $[\![a]\!]_N$ can be derived from the seed seed_N. This is because $a = \sum_{i=1}^{N} [\![a]\!]_i$ is uniformly distributed. The shares $[\![x]\!]_N$ and $[\![Q']\!]_N$ are then computed as

$$[\![x]\!]_N = x - \sum_{i=1}^{N-1} [\![x]\!]_i \quad \text{and} \quad [\![Q']\!]_N = Q' - \sum_{i=1}^{N-1} [\![Q']\!]_i \ .$$

These shares are called the *auxiliary values* of the input sharings. In this paradigm, opening the input shares of the parties in the set $[1 : N] \setminus \{i^*\}$ simply consists in revealing the seeds $\{\mathsf{seed}_i\}_{i \in [1:N] \setminus \{i^*\}}$ and the auxiliary values $([\![x]\!]_N, [\![Q']\!]_N)$ if $i^* \neq N$.

To further enable a compact opening of $N-1$ seeds out of N, we rely on a *tree PRG* (a.k.a. seed tree). The N seeds are generated from a common *root seed* rseed by

$$\{\mathsf{seed}_i\}_{i \in [1:N]} \leftarrow \mathrm{TreePRG}(\mathsf{rseed}) \ .$$

The principle is to label the root of a binary tree of depth $\lceil \log_2 N \rceil$ with $\mathsf{rseed} = \mathsf{seed}_1^{(0)}$. Then, one inductively labels the children of each level-ℓ node with the output of a standard PRG applied to the node's label:

$$\left(\mathsf{seed}_{2i-1}^{(\ell+1)} \parallel \mathsf{seed}_{2i}^{(\ell+1)}\right) \leftarrow \mathrm{PRG}\left(\mathsf{seed}_i^{(\ell)}\right)$$

where the level $\ell = 0$ corresponds to the root and the level $\ell = \log N$ corresponds to the leaves, the N seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_N$ that are used for the shares. To reveal all the seeds but seed_{i^*}, one can reveal the sibling labels of the path from the root rseed to the leaf seed_{i^*}, which we denote:

$$\mathsf{path}_{i^*} \leftarrow \mathrm{GetSiblingPath}(\mathsf{rseed}, i^*) \ .$$

For λ-bit seeds, path_{i^*} is composed of $\log_2 N$ labels of λ bits (assuming N is a power of 2). This means that the seeds $\{\mathsf{seed}_i\}_{i \in [1:N] \setminus \{i^*\}}$ can be opened by sending $\lambda \cdot \log_2 N$ bits instead of $\lambda(N-1)$ bits. The GetSiblingPath routine is detailed in Section 3.2.4.

Using the seed tree optimization the commitments of the shares are done as follows:

- In Step 1 (*Sharing and commitments*), the tree PRG is used the generate the seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_N$. The shares are committed by sending $\mathsf{com}_i = \mathrm{Commit}(\mathsf{seed}_i)$ for every $i \in [1 : N-1]$ and $\mathsf{com}_N = \mathrm{Commit}(\mathsf{seed}_N \parallel [\![x]\!]_N)$ to the verifier.
- In Step 3 (*Hint and commitments*), the prover has received the coefficients $\gamma_1, \ldots, \gamma_m$ and is now able to compute Q'. All the shares $[\![Q']\!]_1, \ldots, [\![Q']\!]_{N-1}$ (which are derived from the seeds) are already committed and the prover only has to compute $[\![Q']\!]_N$ and commit it by sending $\mathsf{com}'_N = \mathrm{Commit}([\![Q']\!]_N)$ to the verifier.

MPCitH optimization: Hypercube party computation. We use the hypercube optimization from [AGH+23] to reduce the number of party computations performed in Step 5 (*MPC simulation*) of the above MPCitH transform, from N down to $\log N + 1$.

The principle is to exploit the linearity of the MPC computation to batch it into group of parties. Denoting $[\![\mathsf{in}]\!]_i$ the input share (witness and hint share) of party i, it computes a broadcast share $[\![\mathsf{broad}]\!]_i = \varphi([\![\mathsf{in}]\!]_i) = [\![\varphi(\mathsf{in})]\!]_i$. We then have that for any set of parties $I \subseteq [1:N]$, we can perform one party computation on $\sum_{i\in I}[\![\mathsf{in}]\!]_i$ to get the sum of broadcast shares $\sum_{i\in I}[\![\mathsf{broad}]\!]_i$. Now let consider a partition $I(0) \cup I(1) = [1:N]$ of the parties. With two batched computations we get two sums of broadcast shares, which sum up to the plain broadcast value

$$\sum_{i\in I(0)} [\![\mathsf{broad}]\!]_i + \sum_{i\in I(1)} [\![\mathsf{broad}]\!]_i = \mathsf{broad} \ . \tag{10}$$

Assume the prover solely sends those two sums of broadcast shares, for some sets $|I(0)| = |I(1)| = N/2$, instead of the N broadcast shares. In case of cheating (*i.e.* at least one party broadcast is inconsistent), the malicious prover is discovered whenever the cheating party is not in the same set as i^* (the non-opened party), which happens with probability $1/2$. Repeating this with another partition $I'(0) \cup I'(1) = [1:N]$, one can obtain a soundness error of $\frac{1}{2} \times \frac{1}{2}$ provided that $I(b) \cap I'(b') = N/4$ for every $b, b' \in \{0,1\}$. Repeating this $\log N$ times, we obtain the hypercube optimization.

To get a working sequence of $D = \log N$ partitions, one can see each party as a vertex $i \equiv (i_1, \ldots, i_D)$ in a hypercube of dimension D (where $(i_1, \ldots, i_D)$ is the binary representation of i). Let us define, for every $d \in [1:D]$ and $b \in \{0,1\}$,

$$I(d,b) = \left\{ i \equiv (i_1, \ldots, i_{d-1}, b, i_{d+1}, \ldots, i_D) \mid i_1, \ldots, i_D \in \{0,1\} \right\} . \tag{11}$$

Each pair $(I(d,0), I(d,1))$ is a partition of the vertices as belonging to two opposite faces of the hypercube. During Step 5 (*MPC simulation*) of the above MPCitH transform, the prover shall only compute and send:

- the sum of broadcast shares $\sum_{i\in I(d,0)}[\![\mathsf{broad}]\!]_i$ for every $d \in [1:D]$,
- the plain broadcast value broad.

This way the prover only performs $D + 1 = \log N + 1$ party computations (one of them being computed on the plain values).

Then in Step 8 (*Verification*) of the MPCitH transform, the verifier checks the consistency of the broadcast shares for every $d \in [1:D]$ as follows:

- If $i^* \in I(d,1)$, the verifier knows all the input shares $[\![\mathsf{in}]\!]_i$ for $i \in I(d,0)$. They recompute the sum $\sum_{i\in I(d,0)}[\![\mathsf{broad}]\!]_i$ by applying the party computation to $\sum_{i\in I(d,0)}[\![\mathsf{in}]\!]_i$, and check its consistency to the sum previously sent by the prover.
- If $i^* \in I(d,0)$, the verifier knows all the input shares $[\![\mathsf{in}]\!]_i$ for $i \in I(d,1)$. They first recompute the sum $\sum_{i\in I(d,1)}[\![\mathsf{broad}]\!]_i$ by applying the party computation to $\sum_{i\in I(d,1)}[\![\mathsf{in}]\!]_i$ and then recover $\sum_{i\in I(d,0)}[\![\mathsf{broad}]\!]_i$ using Equation 10 (since they further received the plain broadcast broad from the prover).

This way the verifier only performs $D = \log N$ party computations.

Parallel repetition. After applying the MPCitH transform to the MQOM MPC protocol we obtain a ZK-PoK for the MQ problem with soundness error

$$\varepsilon = \frac{1}{N} + p\left(1 - \frac{1}{N}\right)$$

where $p = p_1 + (1 - p_1) \cdot p_2$ is the false positive probability given by Theorem 2.1.

In order to scale this to $2^{-\lambda}$ for a target security level λ, we use parallel repetition. This means that the ZK-PoK is repeated τ times in parallel to reach a global soundness error of ε^τ. We stress that a soundness error of ε^τ does not imply an unforgeability of ε^τ once the scheme is made non-interactive using the Fiat-Shamir transform. While fixing the number of repetitions τ to achieve some level of security in the non-interactive setting, one needs to take into account possible forgery attacks such as the one described in Section 5.3.

The Fiat-Shamir transform. The Fiat-Shamir heuristic turns the latter ZK-PoK into the MQOM signature scheme. The principle is to replace the verifier challenge of the ZK-PoK by the outputs of hash functions taking previous prover's communication as input.

Specifically:

1. A first hash h_1 is derived by hashing the share commitments of Step 1 (over all the τ repetitions). This hash is then pseudorandomly expanded into the challenge of Step 2 (different coefficients $\gamma_1, \ldots, \gamma_m$ for the τ repetitions).

2. A second hash h_2 is derived by hashing h_1 together with the hint auxiliary share commitments of Step 3 (over all the τ repetitions). This hash is then pseudorandomly expanded into the challenge of Step 4 (different evaluation points r for the τ repetitions).

3. A third hash h_3 is derived by hashing h_2 together with the broadcast values sent by the prover (over all the τ repetitions). This hash is then pseudorandomly expanded into the challenge of Step 6 (different non-opened party indices i^* for the τ repetitions).

The above hash computations take further inputs:

- *Message:* We introduce the message in the hash computation to obtain a message-binding signature. We choose to introduce the message in the third hash h_3 only. This enables message-independent pre-computation of the Steps 1-to-5, which are the computationally-expansive steps of the signing algorithm.

- *Salt:* For security reasons (*i.e.* to avoid possible collisions), we introduce a salt of 2λ bits. This salt is further passed as argument of the tree PRG (in each node) and the commitments of the shares. The salt is added to the signature to allow the verification.

- *Public key:* We also introduce the public key in the first hash h_1. This makes any forgery attempt specific to a given user, hence preventing adversarial strategies that would invest in heavy precomputation to ease forgery for all (or many of) the users.

2.4.2 Description of the signature scheme

The high-level description of the MQOM signature scheme is wrapped-up in Figure 2. All the pseudorandomness in the MQOM signature scheme is derived from an extendable output hash function (XOF). If the input seed has enough entropy, the output of XOF is assumed to be indistinguishable from true randomness.

INPUT: A secret key $sk := (\mathsf{seed_{eq}}, y, x)$ and a message $m \in \{0,1\}^*$.

Phase 0: Initialization.

1. Expand MQ equations $\{A_i, b_i\}_{i\in[1:m]} \leftarrow \mathrm{XOF}(\mathsf{seed_{eq}})$.
2. Sample a random salt $\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$.
3. Sample a master seed $\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$.
4. Expand mseed as τ root seeds $\{\mathsf{rseed}^{[e]}\}_{e\in[1:\tau]} \leftarrow \mathrm{XOF}(\mathsf{salt}, \mathsf{mseed})$.

Phase 1: Sharing and commitments. For each repetition $e \in [\tau]$,

1. Generate the leaf seeds $\{\mathsf{seed}_i^{[e]}\}_{i\in[1:N]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{rseed}^{[e]})$.
2. Randomly generate the shares $(\llbracket x^{[e]} \rrbracket_i, \llbracket a^{[e]} \rrbracket_i, \llbracket Q'^{[e]} \rrbracket_i) \leftarrow \mathrm{XOF}(\mathsf{salt}, \mathsf{seed}_i^{[e]})$ for $i \in [1 : N-1]$.
3. Compute the last share of the witness $\llbracket x^{[e]} \rrbracket_N = x - \sum_{i=1}^{N-1} \llbracket x^{[e]} \rrbracket_i$.
4. Compute the commitments
   - $\mathsf{com}_i^{[e]} = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{seed}_i^{[e]})$ for $i \in [1 : N-1]$,
   - $\mathsf{com}_N^{[e]} = \mathrm{Commit}(\mathsf{salt}, e, N, \mathsf{seed}_N^{[e]}, \llbracket x^{[e]} \rrbracket_N)$.

Phase 2: First challenge.

1. Compute $h_1 = \mathrm{Hash}_1(pk := (\mathsf{seed_{eq}}, y), \mathsf{salt}, \mathsf{com}_1^{[1]}, \mathsf{com}_2^{[1]}, \ldots, \mathsf{com}_N^{[\tau]},)$.
2. Expand h_1 as the coefficients $\{\gamma_1^{[e]}, \ldots, \gamma_m^{[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_1)$.

Phase 3: Hint and commitments. For each repetition $e \in [\tau]$,

1. Compute the hint $Q'^{[e]}$ from x, $\{A_i, b_i\}_{i\in[1:m]}$, $(\gamma_1^{[e]}, \ldots, \gamma_m^{[e]})$ and $a^{[e]} = \sum_{i=1}^{N} \llbracket a^{[e]} \rrbracket_i$.
2. Compute the last share of the hint $\llbracket Q'^{[e]} \rrbracket_N = Q' - \sum_{i=1}^{N-1} \llbracket Q'^{[e]} \rrbracket_i$.
3. Compute the commitment $\mathsf{com}'^{[e]}_N = \mathrm{Commit}(\mathsf{salt}, e, 0, \llbracket Q'^{[e]} \rrbracket_N)$.

Phase 4: Second challenge.

1. Compute $h_2 = \mathrm{Hash}_2(\mathsf{salt}, h_1, \mathsf{com}'^{[1]}_N, \mathsf{com}'^{[2]}_N, \ldots, \mathsf{com}'^{[\tau]}_N)$.
2. Expand h_2 as the evaluation points $\{r^{[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_2)$.

Phase 5: MPC simulation. For each repetition $e \in [\tau]$,

1. Apply the MPC computation $\mathsf{in}^{[e]} := (x, a^{[e]}, Q'^{[e]}) \mapsto \mathsf{broad}^{[e]} := (\alpha^{[e]}, v^{[e]})$.
2. For $d \in [1 : D]$, apply the MPC computation $\llbracket \mathsf{in}^{[e]} \rrbracket_{I(d,0)} \mapsto \llbracket \mathsf{broad}^{[e]} \rrbracket_{I(d,0)}$ (where $\llbracket \cdot \rrbracket_{I(d,0)} = \sum_{i\in I(d,0)} \llbracket \cdot \rrbracket_i$).

Phase 6: Third challenge.

1. Compute $h_3 = \mathrm{Hash}_3\left(\mathsf{salt}, h_2, \{\mathsf{broad}^{[e]}\}_{e\in[1:\tau]}, \left\{\llbracket \mathsf{broad}^{[e]} \rrbracket_{I(0,d)}\right\}_{d\in[1:D], e\in[1:\tau]}\right)$.
2. Expand h_3 as the non-opened view indices $\{i^{*[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_3)$.

Phase 7: Views opening. For each repetition $e \in [\tau]$,

1. Compute the sibling path of the non-opened seed $\mathsf{path}^{[e]} \leftarrow \mathrm{GetSiblingPath}(\mathsf{salt}, \mathsf{rseed}^{[e]}, i^{*[e]})$.
2. Derive the opened views as
   - $\mathsf{view}^{[e]} = (\mathsf{path}^{[e]}, \llbracket x^{[e]} \rrbracket_N, \llbracket Q'^{[e]} \rrbracket_N)$ if $i^{*[e]} \in [1 : N-1]$,
   - $\mathsf{view}^{[e]} = \mathsf{path}^{[e]}$ if $i^{*[e]} = N$.

Phase 8: Signature.

1. Return the signature $\sigma := \left(\mathsf{salt} \parallel h_1 \parallel h_2 \parallel h_3 \parallel \{\mathsf{view}^{[e]}, \mathsf{broad}^{[e]}, \mathsf{com}^{[e]}_{i^{*[e]}}\}_{e\in[1:\tau]}\right)$.

Figure 2: MQOM – Signing algorithm

INPUT: A public key $sk := (\mathsf{seed_{eq}}, y)$, a message $m \in \{0,1\}^*$ and a signature σ.

Phase 0: Parsing and expansion.

1. $\left(\mathsf{salt} \parallel h_1 \parallel h_2 \parallel h_3 \parallel \{\mathsf{view}^{[e]}, \mathsf{broad}^{[e]}, \mathsf{com}^{[e]}_{i^{*[e]}}\}_{e\in[1:\tau]}\right) \leftarrow \sigma$.
2. Expand MQ equations $\{A_i, b_i\}_{i\in[1:m]} \leftarrow \mathrm{XOF}(\mathsf{seed_{eq}})$.
3. Expand h_1 as the coefficients $\{\gamma_1^{[e]}, \ldots, \gamma_m^{[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_1)$.
4. Expand h_2 as the evaluation points $\{r^{[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_2)$.
5. Expand h_3 as the non-opened view indices $\{i^{*[e]}\}_{e\in[\tau]} \leftarrow \mathrm{XOF}(h_3)$.
6. For each repetition $e \in [\tau]$, parse the opened views as
   - $(\mathsf{path}^{[e]}, [\![x^{[e]}]\!]_N, [\![Q'^{[e]}]\!]_N) = \mathsf{view}^{[e]}$ if $i^{*[e]} \in [1 : N-1]$,
   - $\mathsf{path}^{[e]} = \mathsf{view}^{[e]}$ if $i^{*[e]} = N$.

Phase 1: Recomputing shares and commitments. For each repetition $e \in [\tau]$,

1. Recover the open seeds from the sibling path $\{\mathsf{seed}_i^{[e]}\}_{i\in[1:N]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{path}^{[e]})$.
2. For $i \in [1 : N-1] \setminus \{i^{*[e]}\}$,
   - Regenerate the shares $([\![x^{[e]}]\!]_i, [\![a^{[e]}]\!]_i, [\![Q'^{[e]}]\!]_i) \leftarrow \mathrm{XOF}(\mathsf{salt}, \mathsf{seed}_i^{[e]})$ for $i \in [1 : N-1]$.
   - Recompute the commitments $\mathsf{com}_i^{[e]} = \mathrm{Commit}(\mathsf{salt}, e, i, \mathsf{seed}_i^{[e]})$.
3. If $i^{*[e]} \neq N$,
   - Regenerate the shares $[\![a^{[e]}]\!]_N \leftarrow \mathrm{XOF}(\mathsf{salt}, \mathsf{seed}_N^{[e]})$.
   - Recompute the commitment $\mathsf{com}_N^{[e]} = \mathrm{Commit}(\mathsf{salt}, e, N, \mathsf{seed}_N^{[e]})$.
   - Recompute the commitment $\mathsf{com}'^{[e]}_N = \mathrm{Commit}(\mathsf{salt}, e, 0, [\![Q'^{[e]}]\!]_N)$.

   NB: If one of these commitments is incorrect, the recomputed hashes (h'_1, h'_2) won't match (h_1, h_2) in Phase 3 below.

Phase 2: MPC simulation. For each repetition $e \in [\tau]$,

1. For $d \in [1 : D]$, s.t. $i^{*[e]} \in I(1, d)$,
   - Apply the MPC computation $[\![\mathsf{in}^{[e]}]\!]_{I(d,0)} \mapsto [\![\mathsf{broad}^{[e]}]\!]_{I(d,0)}$.
2. For $d \in [1 : D]$, s.t. $i^{*[e]} \in I(d, 0)$,
   - Apply the MPC computation $[\![\mathsf{in}^{[e]}]\!]_{I(d,1)} \mapsto [\![\mathsf{broad}^{[e]}]\!]_{I(d,1)}$.
   - Recover $[\![\mathsf{broad}^{[e]}]\!]_{I(d,0)} = \mathsf{broad}^{[e]} - [\![\mathsf{broad}^{[e]}]\!]_{I(d,1)}$.

   with $[\![\cdot]\!]_{I(d,b)} = \sum_{i\in I(b,d)} [\![\cdot]\!]_i$, $[\![\mathsf{in}^{[e]}]\!]_i := ([\![x]\!]_i, [\![a^{[e]}]\!]_i, [\![Q'^{[e]}]\!]_i)$, $[\![\mathsf{broad}^{[e]}]\!]_i := ([\![\alpha^{[e]}]\!]_i, [\![v^{[e]}]\!]_i)$.

   NB: If one of these sum of broadcast shares is incorrect, the recomputed hash h'_3 won't match h_3 in Phase 3 below.

Phase 3: Recomputing hashes.

1. Compute $h'_1 = \mathrm{Hash}_1(pk := (\mathsf{seed_{eq}}, y), \mathsf{salt}, \mathsf{com}_1^{[1]}, \mathsf{com}_2^{[1]}, \ldots, \mathsf{com}_N^{[\tau]},)$.
2. Compute $h'_2 = \mathrm{Hash}_2(\mathsf{salt}, h_1, \mathsf{com}'^{[1]}_N, \mathsf{com}'^{[2]}_N, \ldots, \mathsf{com}'^{[\tau]}_N)$.
3. Compute $h'_3 = \mathrm{Hash}_3\left(\mathsf{salt}, h_2, \{\mathsf{broad}^{[e]}\}_{e\in[1:\tau]}, \left\{[\![\mathsf{broad}^{[e]}]\!]_{I(d,0)}\right\}_{d\in[1:D], e\in[1:\tau]}\right)$.

Phase 4: Verification.

1. If $\mathsf{broad} = (\alpha_1, \ldots, \alpha_{n_2}, v)$ with $v \neq 0$, then return REJECT.
2. If $(h_1, h_2, h_3) \neq (h'_1, h'_2, h'_3)$, then return REJECT.
3. Else return ACCEPT.

Figure 3: MQOM – Verification algorithm

3 Detailed algorithmic description

3.1 Notations

The elements manipulated by the signature and verification algorithms are vectors of field elements. In the following, we denote $\mathbb{F}$ to mean a field which might be the SD base field $\mathbb{F}_q$ or the extension field $\mathbb{F}_{q^\eta}$.

Vectors. For a vector v over $\mathbb{F}$, we denote its length $|v|$, namely $v \in \mathbb{F}^\ell \Leftrightarrow |v| = \ell$. We further denote $v\,[i]$ the ith coordinate of v. For any two vectors v_1 and v_2, we denote $(v_1 \parallel v_2) \in \mathbb{F}^{|v_1|+|v_2|}$ their concatenation. For a vector $v \in \mathbb{F}^{|v|}$, for any $n \in \mathbb{N}$, and for any sequence of positive integers $\ell_1, \ldots, \ell_n$ such that $|v| = \ell_1 + \cdots + \ell_n$, we denote

$$(v_1, \ldots, v_n) \leftarrow \mathrm{Parse}(v, \mathbb{F}^{\ell_1}, \ldots, \mathbb{F}^{\ell_n})$$

the operation which splits v into n vectors of field elements such that

$$v = (v_1 \parallel \ldots \parallel v_n) \quad \text{and} \quad |v_i| = \ell_i \ \forall i \in [1:n] \ .$$

We shall also manipulate two-dimensional vectors of field elements. For instance $v \in (\mathbb{F}^\ell)^d$ is a vector with d coordinates which are vectors of $\mathbb{F}^\ell$. For such a vector, we naturally extend the coordinate notation such that $v\,[i] \in \mathbb{F}^\ell$ is the ith coordinate of v, itself a vector, and $v\,[i]\,[j] \in \mathbb{F}$ is the jth coordinate of $v\,[i]$. We further extend the definition of the Parse function to handle two-dimensional vectors. For some vector $v \in \mathbb{F}^{|v|}$, if we denote

$$(v_1, \ldots, v_n) \leftarrow \mathrm{Parse}(v, (\mathbb{F}^{\ell_1})^{d_1}, \ldots, (\mathbb{F}^{\ell_n})^{d_n})$$

then v_k is the two-dimensional vector from $(\mathbb{F}^{\ell_k})^{d_k}$ satisfying

$$v_k\,[i]\,[j] = v\,[\delta_k + i \cdot \ell_k + j] \quad \forall (i,j) \in [1:d_k] \times [1:\ell_k] \ .$$

where $\delta_1 = 0$ and $\delta_k = \ell_1 d_1 + \cdots + \ell_{k-1} d_{k-1}$ for $k > 1$.

Serialized variables. For any tuple $(v_1, \ldots, v_n)$ of two-dimensional vectors, the Serialize function "flattens" this tuple by returning the vector v defined as:

$$v = \mathrm{Serialize}(v_1, \ldots, v_n) \in \mathbb{F}^{\ell_1 d_1 + \cdots + \ell_n d_n}$$
$$\Leftrightarrow \quad (v_1, \ldots, v_n) = \mathrm{Parse}(v, (\mathbb{F}^{\ell_1})^{d_1}, \ldots, (\mathbb{F}^{\ell_n})^{d_n}) \ .$$

In the implementation, the output of Serialize are represented in a compact way, as follows:

- For $\mathbb{F} = \mathbb{F}_{31}$, each coordinate of an $\mathbb{F}_{31}$-vector output by Serialize is represented on 5 bits. Each chunk of 8 elements e_1, …, e_8 in a serialized variable is hence stored on 5 bytes B_1, …, B_5 such that
$$(e_1 \parallel \cdots \parallel e_8) = (B_1 \parallel \cdots \parallel B_5)$$
The last chunk of a serialized $\mathbb{F}_{31}$-vector might be stored on a smaller number of bytes (specifically on $\lceil \ell * 5/8 \rceil$ bytes where ℓ is the number of coordinates in the last chunk).
- For $\mathbb{F} = \mathbb{F}_{251}$, each coordinate of an $\mathbb{F}_{251}$-vector output by Serialize is stored on a single byte.

In the algorithmic description below, we sometimes perform linear operations between serialized variables, such as $\mathsf{var1} + \mathsf{var2}$ (or $\mathsf{var1} - \mathsf{var2}$). This is to be interpreted as adding (or subtracting) each coordinate of the $\mathbb{F}$-vector represented by the serialized variable, while keeping the compact representation for the serialized output.

Arithmetic operations. In the algorithmic description, we shall use the operator $\cdot$ to denote the product over $\mathbb{F}_q$. We shall further use this operator for the scalar product between a value $u \in \mathbb{F}_q$ and a vector $v = (v_1, \ldots, v_\ell) \in \mathbb{F}_q^\ell$, that is

$$u \cdot v = (u \cdot v[1], \ldots, u \cdot v[\ell]) \ .$$

An element of $\mathbb{F}_{q^\eta}$ is represented as a vector of η elements of $\mathbb{F}_q$. For any $v \in \mathbb{F}_{q^\eta}$, we denote $v \equiv (v_1, \ldots, v_\eta) \in \mathbb{F}_q^\eta$ the relation between v and the corresponding $\mathbb{F}_q$-vector. We shall use the operator $\otimes$ to denote the product over $\mathbb{F}_{q^\eta}$. For any $u, v \in \mathbb{F}_q^\eta$, with $u \equiv (u_1, \ldots, u_\eta)$ and $v \equiv (v_1, \ldots, v_\eta)$, the product $z = u \otimes v$ is defined as:

$$z \ \equiv \ (z_1, \ldots, z_\eta) \quad \text{s.t.} \quad \sum_{i=1}^{\eta} z_i u^{i-1} \ = \ \left(\sum_{i=1}^{\eta} u_i u^{i-1}\right)\left(\sum_{i=1}^{\eta} v_i u^{i-1}\right) \bmod f(u) \ , \quad (12)$$

where $f(u)$ is the degree-η irreducible polynomial of $\mathbb{F}_q[u]$ such that $\mathbb{F}_{q^\eta} \equiv \mathbb{F}_q[u]/f(u)$.

Intermediate variables. We use the mathematical notations introduced in Section 2 and summarized in Table 1. The different seeds, indexes and serialized variables are summarized in Table 2.

Table 2: Descriptions of the low-level notations used in our scheme.

Seeds:		
$\mathsf{seed}_{\mathsf{root}}$	$\{0,1\}^\lambda$	Root seed which is expanded into $\mathsf{seed}_{\mathsf{x}}$ and $\mathsf{seed}_{\mathsf{eq}}$.
$\mathsf{seed}_{\mathsf{x}}$	$\{0,1\}^\lambda$	Seed for the generation of the MQ solution x.
$\mathsf{seed}_{\mathsf{eq}}$	$\{0,1\}^\lambda$	Seed for the generation of the MQ equations $\{A_i, b_i\}$.
mseed	$\{0,1\}^\lambda$	Master seed for all the pseudo-randomness of the signature.
salt	$\{0,1\}^{2\lambda}$	Salt for the pseudo-randomness and commitments of the signature.
rseed$[e]$	$\{0,1\}^\lambda$	Seeds which are the roots of the seed tree.
seed$[e][i]$	$\{0,1\}^\lambda$	Parties' seeds (leaves of the seed trees).
Indexes:		
e	$1, \ldots, \tau$	Index for the current repetition.
i	$1, \ldots, N$	Index for the current party.
d	$1, \ldots, D$	Index for the dimension of the hypercube.
j	$1, \ldots, n_2$	Index for the current chunk.
k	$1, \ldots, n_1$	Position in a chunk.
Serialized variables:		
x	$\mathbb{F}_q^n$	Serialized plain MQ solution x.
y	$\mathbb{F}_q^m$	Serialized plain MQ output y.
unif_plain	$\mathbb{F}_q^{\eta n_2}$	Serialized plain uniformly-sampled $a = (a[1], \ldots, a[n_2])$.
hint_plain	$\mathbb{F}_q^{\eta(2n_1-1)}$	Serialized hint polynomial Q'.
in_plain	$\mathbb{F}_q^{n+\eta n_2+\eta(2n_1-1)}$	Serialized plain input: in_plain $= (\mathsf{x}, \mathsf{unif\_plain}, \mathsf{hint\_plain})$.
broad_plain	$\mathbb{F}_q^{\eta n_2}$	Serialized plain broadcast: $\alpha = (\alpha[1], \ldots, \alpha[n_2])$.
in_mshare	$\mathbb{F}_q^{n+\eta n_2+\eta(2n_1-1)}$	Serialized input main share: $[\![x]\!]_{I(d,b)}, [\![a]\!]_{I(d,b)}, [\![Q']\!]_{I(d,b)}$.
broad_mshare	$\mathbb{F}_q^{\eta(n_2+1)}$	Serialized broadcast main share: $[\![\alpha]\!]_{I(d,b)}, [\![v]\!]_{I(d,b)}$.

3.2 Subroutines

In this subsection, we describe different subroutines which are involved in our key generation, signature, and verification algorithms. These sub-routines are related to (i) the MPC simulation, (ii) the randomness generation, (iii) the hash functions, and (iv) the seed trees.

3.2.1 MPC subroutines

We describe hereafter all the subroutines required for the MPC simulation following the description of Section 2.3.

Polynomial evaluation. We define the function PolyEval which takes as input an $\mathbb{F}_{q^\eta}$-vector Q representing the coefficients of polynomial of $\mathbb{F}_{q^\eta}[u]$ and a point $r \in \mathbb{F}_{q^\eta}$, computes the evaluation $Q(r)$. Formally, we have

$$\text{PolyEval} : \begin{cases} \bigcup_d (\mathbb{F}_{q^\eta})^d \times \mathbb{F}_{q^\eta} & \to \mathbb{F}_{q^\eta} \\ (Q, r) & \mapsto \sum_{i=1}^{|Q|} Q\,[i] \cdot r^{i-1} \end{cases} \qquad \text{where } r^{i-1} = \underbrace{r \otimes r \otimes \cdots \otimes r}_{i-1 \text{ times}} .$$

Polynomial product. We define the function PolyProduct which takes as input two $\mathbb{F}_{q^\eta}$-vectors P_1 and P_2 representing the coefficients of polynomials from $\mathbb{F}_{q^\eta}[u]$ and output a $\mathbb{F}_{q^\eta}$-vector Q of length $|Q| = |P_1| + |P_2| - 1$ representing their product. Formally, we have

$$\text{PolyProduct} : \begin{cases} \bigcup_d (\mathbb{F}_{q^\eta})^d \times \bigcup_d (\mathbb{F}_{q^\eta})^d & \to \mathbb{F}_{q^\eta} \\ (P_1, P_2) & \mapsto Q \end{cases} \qquad \text{where } Q(u) = P_1(u) \cdot P_2(u) .$$

Polynomial interpolation. We define two subroutines for polynomial interpolation. The subroutine InterpolateX maps a vector $x \in \mathbb{F}_q^n$ to n_2 polynomials $X\,[1]$, …, $X\,[n_2]$ represented as vectors of coefficients in $\mathbb{F}_q^{n_1}$. These polynomials are computed by interpolating each n_1-long chunk of x (where the last chunk is possibly padded with 0's) on the set $\{f_1, \ldots, f_{n_1}\}$ as depicted by Equation 2 and Equation 6.

The subroutine InterpolateW works the same way as InterpolateX with two differences: (1) it works over the extension field $\mathbb{F}_{q^\eta}$, (2) the interpolation of the last chunk polynomial uses $n_1' \leq n_1$ elements. Specifically, InterpolateW maps a vector $w \in \mathbb{F}_{q^\eta}^n$ to n_2 polynomials $W\,[1]$, …, $W\,[n_2]$. These polynomials are computed by interpolating the $n_2 - 1$ first n_1-long chunks of w on the set $\{f_1, \ldots, f_{n_1}\}$ and the last n_1'-long chunk on the set $\{f_1, \ldots, f_{n_1'}\}$ as depicted by Equation 2 and Equation 6.

Plain hint computation. The subroutine ComputePlainHint, described in Algorithm 1, computes the plain value of the hint Q'. It takes as input the witness x (or equivalently the polynomials $X\,[j]$), the random masks $a\,[1]$, …, $a\,[n_2]$, the random coefficients γ_1, …, γ_m, and the MQ equations $\{A_i, b_i\}_i$, from which it first computes w (or equivalently the polynomials $W\,[j]$) and then the hint polynomial Q' (plain value) according to Equation 8 and Equation 4.

The subroutine ComputePlainHint makes use the function TruncateQ which returns the $\mathbb{F}_q^{n_1-2}$ vector Q' obtained by dropping the left-most element of its input $Q \in \mathbb{F}_q^{n_1-1}$. In polynomial terms, this means computing $Q'(u) = (Q(u) - q_0)/u$, where q_0 denotes the constant term of Q.

Algorithm 1 ComputePlainHint

Input: x, $\mathsf{unif\_plain}$, chal_1, $\{A_i, b_i\}_i$
Output: $\mathsf{hint\_plain}$

$x \leftarrow \mathrm{Parse}(\mathsf{x}, \mathbb{F}_q^n)$
$a \leftarrow \mathrm{Parse}(\mathsf{unif\_plain}, \mathbb{F}_{q^\eta}^{n_2})$
$(\gamma_1, \ldots, \gamma_m) \leftarrow \mathrm{Parse}(\mathsf{chal}_1, \mathbb{F}_{q^\eta}^m)$
$w \leftarrow (\sum_{i=1}^m \gamma_i A_i)x$ ▷ $w \in \mathbb{F}_{q^\eta}^n$
$X \leftarrow \mathrm{InterpolateX}(x)$
$W \leftarrow \mathrm{InterpolateW}(w)$
$Q \leftarrow \sum_{j=1}^{n_2} \mathrm{PolyProduct}(W[j] + a[j] \cdot V, X[j])$ ▷ $Q \in \mathbb{F}_{q^\eta}^{2n_1}$
$Q' \leftarrow \mathrm{TruncateQ}(Q)$ ▷ $Q' \in \mathbb{F}_{q^\eta}^{2n_1-1}$
$\mathsf{hint\_plain} = \mathrm{Serialize}(Q')$
return $\mathsf{hint\_plain}$

Computation of plain broadcast values. The subroutine ComputePlainBroadcast, described in Algorithm 2, computes the publicly recomputed values from the broadcast shares during the MPC protocol, namely the evaluations $\alpha[j] = \tilde{W}[1](r), \ldots, \alpha[n_2] = \tilde{W}[1](r)$. It takes as input the witness x (or equivalently the polynomials $X[j]$), the random masks $a[1], \ldots, a[n_2]$, the random coefficients $\gamma_1, \ldots, \gamma_m$, the random evaluation point r, and the MQ equations $\{A_i, b_i\}_i$, from which it first computes w (or equivalently the polynomials $W[j]$) and then the evaluations $\alpha[j] = \tilde{W}[j](r)$.

Algorithm 2 ComputePlainBroadcast

Input: $\mathsf{in\_plain} := (\mathsf{x}, \mathsf{unif\_plain}, \mathsf{hint\_plain})$, chal_1, chal_2, $\{A_i, b_i\}_i$
Output: $\mathsf{broad\_plain}$

$x \leftarrow \mathrm{Parse}(\mathsf{x}, \mathbb{F}_q^n)$
$a \leftarrow \mathrm{Parse}(\mathsf{unif\_plain}, \mathbb{F}_{q^\eta}^{n_2})$
$(\gamma_1, \ldots, \gamma_m) \leftarrow \mathrm{Parse}(\mathsf{chal}_1, \mathbb{F}_{q^\eta}^m)$
$r \leftarrow \mathrm{Parse}(\mathsf{chal}_2, \mathbb{F}_{q^\eta})$
$w \leftarrow (\sum_{i=1}^m \gamma_i A_i)x$ ▷ $w \in \mathbb{F}_{q^\eta}^n$
$W \leftarrow \mathrm{InterpolateW}(w)$
for $j \in [1 : n_2]$ **do**
    $\tilde{W}[j] \leftarrow W[j] + a[j] \cdot V$
    $\alpha[j] \leftarrow \mathrm{PolyEval}(\tilde{W}[j], r)$
$\mathsf{broad\_plain} = \mathrm{Serialize}(\alpha)$
return $\mathsf{broad\_plain}$

Remark 1. *The plain value $v = 0$ is omitted from $\mathsf{broad\_plain}$. In the verification algorithm, while subtracting a sum of broadcast shares to the plain broadcast value, the latter must hence be padded with η 0's to match the format of shared broadcast values. See Remark 3 below.*

Computation of broadcast shares. The subroutine PartyComputation, described in Algorithm 3, performs the computation of a main party. Namely from the input shares of the main party, it computes the corresponding broadcast shares. For some $d \in [1 : D]$ and $b \in \{0, 1\}$, it computes the sum of broadcast shares

$$[\![\mathsf{broad}^{[e]}]\!]_{I(d,b)} := ([\![\alpha^{[e]}]\!]_{I(d,b)}, [\![v^{[e]}]\!]_{I(d,b)})$$

from the sum of input shares

$$[\![\mathsf{in}^{[e]}]\!]_{I(d,b)} := ([\![x^{[e]}]\!]_{I(d,b)}, [\![a^{[e]}]\!]_{I(d,b)}, [\![Q'^{[e]}]\!]_{I(d,b)})$$

where $[\![\cdot]\!]_{I(d,b)} = \sum_{i \in I(d,b)} [\![\cdot]\!]_i$ and $I(d,b)$ is as defined in Equation 11.

This subroutine takes as input the sum of input shares $[\![\mathsf{in}^{[e]}]\!]_{I(d,b)}$ (argument in_mshare), the challenge random coefficients $\gamma_1, \ldots, \gamma_m$ (argument chal_1), the evaluation point r (argument chal_2), and the MQ instance $(\{A_i, b_i\}_i, y)$, from which it computes the sum of broadcast shares $[\![\mathsf{broad}^{[e]}]\!]_{I(d,b)}$.

This subroutine further takes as input a Boolean with_offset which is `True` if $d = 1$ and `False` otherwise. This Boolean indicates whether the constant part of the computed affine function should be introduced or not.

Algorithm 3 PartyComputation

Input: in_mshare, chal_1, chal_2, $(\{A_i, b_i\}_i, \mathsf{y})$, broad_plain, with_offset
Output: broad_mshare

1: $(\mathsf{x\_mshare}, \mathsf{unif\_mshare}, \mathsf{hint\_mshare}) \leftarrow \mathsf{in\_mshare}$
2: $y \leftarrow \mathrm{Parse}(\mathsf{y}, \mathbb{F}_q^m)$
3: $[\![x]\!]_{I(d,b)} \leftarrow \mathrm{Parse}(\mathsf{x\_mshare}, \mathbb{F}_q^n)$
4: $[\![a]\!]_{I(d,b)} \leftarrow \mathrm{Parse}(\mathsf{unif\_mshare}, \mathbb{F}_{q^\eta}^{n_2})$
5: $[\![Q']\!]_{I(d,b)} \leftarrow \mathrm{Parse}(\mathsf{hint\_mshare}, \mathbb{F}_{q^\eta}^{2n_1-1})$
6: $(\gamma_1, \ldots, \gamma_m) \leftarrow \mathrm{Parse}(\mathsf{chal}_1, \mathbb{F}_{q^\eta}^m)$
7: $r \leftarrow \mathrm{Parse}(\mathsf{chal}_2, \mathbb{F}_{q^\eta})$
8: $\alpha \leftarrow \mathrm{Parse}(\mathsf{broad\_plain}, \mathbb{F}_{q^\eta})$
9: $[\![w]\!]_{I(d,b)} \leftarrow (\sum_{i=1}^m \gamma_i A_i)[\![x]\!]_{I(d,b)}$ ▷ $[\![w]\!]_{I(d,b)} \in \mathbb{F}_{q^\eta}^n$
10: **if** with_offset is `True` **then**
11: $\quad [\![z]\!]_{I(d,b)} \leftarrow \sum_{i=1}^m \gamma_i (y_i - b_i^T [\![x]\!]_{I(d,b)})$ ▷ $[\![z]\!]_{I(d,b)} \in \mathbb{F}_{q^\eta}$
12: **else**
13: $\quad [\![z]\!]_{I(d,b)} \leftarrow -\sum_{i=1}^m \gamma_i \cdot b_i^T [\![x]\!]_{I(d,b)}$
14: $[\![X]\!]_{I(d,b)} \leftarrow \mathrm{InterpolateX}([\![x]\!]_{I(d,b)})$
15: $[\![W]\!]_{I(d,b)} \leftarrow \mathrm{InterpolateW}([\![w]\!]_{I(d,b)})$
16: **for** $j \in [1 : n_2]$ **do**
17: $\quad [\![\tilde{W}\,[j]]\!]_{I(d,b)} \leftarrow [\![W\,[j]]\!]_{I(d,b)} + [\![a\,[j]]\!]_{I(d,b)} \cdot V$
18: $\quad [\![\alpha\,[j]]\!]_{I(d,b)} \leftarrow \mathrm{PolyEval}([\![\tilde{W}\,[j]]\!]_{I(d,b)}, r)$
19: $[\![q_0]\!] = (n_1)^{-1}\Big([\![z]\!] - \sum_{i=1}^{n_1} f_i \cdot \mathrm{PolyEval}([\![Q']\!]_{I(d,b)}, f_i)\Big)$
20: $[\![v]\!]_{I(d,b)} = r \cdot \mathrm{PolyEval}([\![Q']\!]_{I(d,b)}, r) + [\![q_0]\!]_{I(d,b)} - \sum_{j=1}^{n_2} \alpha\,[j] \cdot \mathrm{PolyEval}([\![X\,[j]]\!]_{I(d,b)}, r)$
21: $\mathsf{broad\_mshare} = \mathrm{Serialize}([\![\alpha]\!]_{I(d,b)}, [\![v]\!]_{I(d,b)})$
22: **return** broad_mshare

3.2.2 Pseudorandomness generation

Several subroutines used in the MQOM signature schemes involve pseudorandomness generation from a seed. Several seeds are expanded from a master seed in the key generation and the signature algorithms. One also needs to sample pseudorandom sequences of field elements from a seed to expand the MQ instance and the sharings involved in the signature and verification algorithms. Finally, pseudorandomness generation is also involved to derive the challenges (MPC challenges and view-opening challenge) from the Fiat-Shamir hashes h_1, h_2 and h_3.

Extendable output function. The pseudorandomness in MQOM is generated through an extendable output hash function (XOF). Such a function takes an arbitrary-long input bitstring $\mathsf{in} \in \{0,1\}^*$ and produces an arbitrary-long output bitstring $\mathsf{out} \in \{0,1\}^*$ whose length is tailored to the requirements of the application. Formally, a XOF is equipped with two routines: XOF.Init(in) initializes the XOF state with the input $\mathsf{in} \in \{0,1\}^*$. Once initialized, the XOF can be queried with the routine XOF.GetByte() to generate the next byte of the output out associated to in. The concrete instance of the XOF we use in the MQOM scheme is given in Section 4.6. In our context, we use the XOF as a secure pseudorandom generator (PRG) which tolerates input seeds of variable lengths.

Sampling from XOF. We shall denote by Sample, the routine generating pseudorandom element from an arbitrary set $\mathcal{V}$. A call to

$$v \leftarrow \text{XOF.Sample}(\mathcal{V})$$

outputs a uniform random element $v \in \mathcal{V}$. The Sample routine relies on calls to GetByte to generate pseudorandom bytes which are then formatted to obtain a uniform variable $v \in \mathcal{V}$, possibly using rejection sampling. The implementation of Sample depends on the target set $\mathcal{V}$. We detail the case of sampling field elements hereafter, namely when $\mathcal{V} = \mathbb{F}_q^n$ for some n.

Sampling field elements. The subroutine XOF. SampleFieldElements(n) samples n pseudorandom elements from $\mathbb{F}_q$. It assumes that the XOF has been previously initialized by a call to XOF.Init($\cdot$). We consider two different fields: $\mathbb{F}_{31}$ and $\mathbb{F}_{251}$.

- For $\mathbb{F}_{31}$, we use XOF.GetByte() to generate bytes five-by-five. From 5 bytes, we get 8 chunks of 5 bits, eaching yielding an element of $\mathbb{F}_{31}$, with a $1/32$ rejection rate. Specifically, the SampleFieldElements routine works as follows:

  $i = 1$
  while $i \leq n$ **do**
  for $j = 1$ **to** 5 **do**
  $B_j \leftarrow$ XOF.GetByte()
  $(e_1 \parallel \cdots \parallel e_8) \leftarrow (B_1 \parallel \cdots \parallel B_5)$
  for $j = 1$ **to** 8 **do**
  if $e_j \neq 31$ **then**
  $f_i = e_j$; i ++
  return $(f_1, \ldots, f_n)$

  Remark 2. *In the instruction* $(e_1 \parallel \cdots \parallel e_8) \leftarrow (B_1 \parallel \cdots \parallel B_5)$*, we consider that the bits of* e_i *and* B_i *are represented in little-endian when concatenating. It implies that the*

least-significant bit of e_1 is the least-significant bit of B_1 and that the most-significant bit of e_8 is the most-significant bit of B_5.

- For $\mathbb{F}_{251}$, we use XOF.GetByte() to generate each element of the sequence, with a 5/256 rejection rate. Specifically, the SampleFieldElements routine works as follows:

  1: $i = 1$
  2: **while** $i \leq n$ **do**
  3: $\quad B \leftarrow$ XOF.GetByte()
  4: $\quad$ **if** $B \in \{0, 1, \ldots, 250\}$ **then**
  5: $\quad\quad f_i = B$; i ++
  6: **return** $(f_1, \ldots, f_n)$

Seed expansion. The subroutine ExpandSeed expands a salt and a master seed into a given number of seeds. Specifically, a call to ExpandSeed($\mathsf{salt}, \mathsf{seed}, n$) initializes the XOF by calling XOF.Init($\mathsf{salt} \parallel \mathsf{seed}$) and then calls XOF.GetByte() to generate a stream of bytes B_1, ..., $B_{n\lambda/8}$ which are divided into n output λ-bit seeds seed_1, ..., seed_n as follows:

$$(\underbrace{B_1, \ldots, B_{\lambda/8}}_{\mathsf{seed}_1}, \ldots, \underbrace{B_{(n-1)\lambda/8+1}, \ldots, B_{n\lambda/8}}_{\mathsf{seed}_n})$$

Expansion of the MQ secret. The subroutine ExpandSecret takes as input λ-bit seed seed_x and returns a vectors x. A call to ExpandSecret(seed_x) performs as follows:

1: XOF.Init(seed_x)
2: $x \leftarrow$ XOF. SampleFieldElements(n)
3: **return** x

Expansion of the MQ equations. The subroutine ExpandMQEquations takes as input λ-bit seed $\mathsf{seed}_\mathsf{eq}$ and returns m triangular matrices $A_1, \ldots, A_m \in \mathbb{F}_q^{n \times n}$ and m vectors $b_1, \ldots, b_m \in \mathbb{F}_q^n$. A call to ExpandMQEquations($\mathsf{seed}_\mathsf{eq}$) performs as follows:

1: XOF.Init($\mathsf{seed}_\mathsf{eq}$)
2: **for** $i = 1$ **to** m **do**
3: $\quad$ **for** $j = 1$ **to** n **do**
4: $\quad\quad (a_{j,1}, \ldots, a_{j,j}) \leftarrow$ XOF. SampleFieldElements(j)
5: $\quad\quad (a_{j,j+1}, \ldots, a_{j,n}) \leftarrow (0, \ldots, 0)$
6: $\quad A_i = (a_{j,k})_{1 \leq j \leq n, 1 \leq k \leq n}$
7: $\quad b_i \leftarrow$ XOF. SampleFieldElements(n)
8: **return** $\{A_i, b_i\}_i$

Expansion of MPC challenge. The subroutines ExpandFirstMPCChallenge expands the first Fiat-Shamir hash h_1 into the sequences of coefficients $\gamma_1, \ldots, \gamma_m \in \mathbb{F}_{q^\eta}$ (one per execution). For each execution, the challenge is sampled as a sequence of $m\eta$ field elements. A call to ExpandFirstMPCChallenge(h_1) performs as follows:

1: XOF.Init(h_1)
2: **for** $e = 1$ **to** τ **do**
3: $\quad \mathsf{chal}_1[e] \leftarrow$ XOF. SampleFieldElements($m\eta$)
4: **return** $\{\mathsf{chal}_1[e]\}_{e \in [1:\tau]}$

The subroutines ExpandSecondMPCChallenge expands the first Fiat-Shamir hash h_2 into the evaluation points (one per execution). For each execution, the evaluation point $r \in \mathbb{F}_{q^\eta}$ is sampled as a sequence of η field elements. A call to ExpandSecondMPCChallenge(h_2) performs as follows:

1: XOF.Init(h_2)
2: **for** $e = 1$ **to** τ **do**
3: $\quad \mathsf{chal}_2[e] \leftarrow$ XOF.SampleFieldElements(η)
4: $\quad$ **while** $\mathsf{chal}_2[e] \in \{f_1, \ldots, f_{n_1}\}$ **do**
5: $\quad\quad \mathsf{chal}_2[e] \leftarrow$ XOF.SampleFieldElements(η)
6: **return** $\{\mathsf{chal}_2[e]\}_{e\in[1:\tau]}$

Expansion of view-opening challenge. The subroutine ExpandViewChallenge, expands the third Fiat-Shamir hash h_3 into the view-opening challenge $i^*[1], \ldots, i^*[\tau]$, where $i^*[e] \in [1 : N]$ is the index of the non-opened party for execution e. This subroutine assumes that $N = 2^D$ (N is a power of 2) with $D \leq 16$. A call to ExpandViewChallenge(h_3) performs as follows:

1: XOF.Init(h_3)
2: **for** $e = 1$ **to** τ **do**
3: $\quad B_0 \leftarrow$ XOF.GetByte()
4: $\quad B_1 \leftarrow$ XOF.GetByte()
5: $\quad i^*[e] = (B_0 + 256 \cdot B_1) \mod 2^D$
6: **return** $\{i^*[e]\}_{e\in[1:\tau]}$

3.2.3 Hashing and commitments

Several subroutines used in the MQOM signature scheme involve cryptographic hashing. This is the case of the subroutines computing the Fiat-Shamir hashes, the commitments and the seed trees.

Cryptographic hash function. The different subroutines all use a common cryptographic hash function

$$\text{Hash} : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda} .$$

The concrete instance of the hash function we use in the MQOM scheme is given in Section 4.6.

We use domain separation for the different usages of the hash function. This is simply done by prepending a fixed byte value to the data to be hashed, as specified below for the different cases.

Commitments. The subroutine Commit takes as input a 2λ-bit salt, an execution index e, a share index i and some data $\mathsf{data} \in \{0,1\}^*$. It hashes them all together and returns the corresponding digest. Specifically, we define:

$$\text{Commit}(\mathsf{salt}, e, i, \mathsf{data}) = \text{Hash}(0 \parallel \mathsf{salt} \parallel e_0 \parallel e_1 \parallel i_0 \parallel i_1 \parallel \mathsf{data}) ,$$

where e_0, e_1, i_0, i_1 are the byte values such that $e = e_0 + 256 \cdot e_1$ and $i = i_0 = 256 \cdot i_1$, where 0, e_0, e_1, i_0 and i_1 are encoded on one byte, and where salt is encoded on $2\lambda/8$ bytes.

Fiat-Shamir Hashes. The hash functions Hash_1, Hash_2, Hash_3 used to derive the Fiat-Shamir Hashes h_1, h_2 and h_3 are defined as:

$$\begin{aligned}\mathrm{Hash}_1(\mathsf{data}) &= \mathrm{Hash}(1 \parallel \mathsf{data})\\ \mathrm{Hash}_2(\mathsf{data}) &= \mathrm{Hash}(2 \parallel \mathsf{data})\\ \mathrm{Hash}_3(\mathsf{data}) &= \mathrm{Hash}(3 \parallel \mathsf{data})\end{aligned}$$

where the prefixes 1, 2 and 3 are encoded on one byte.

Hashes in seed trees. The pseudorandom generation occuring in each node of a seed tree relies on a call to a hash function. The hash function Hash_4 used in the seed trees (see description below) is defined as:

$$\mathrm{Hash}_4(\mathsf{data}) = \mathrm{Hash}(4 \parallel \mathsf{data}) \ ,$$

where the prefix 4 is encoded on one byte.

As described hereafter, the child nodes in a seed trees are derived with one call to Hash_4 as follows:

$$(\mathsf{nodes}[2i] \parallel \mathsf{nodes}[2i+1]) = \mathrm{Hash}_4(\mathsf{salt} \parallel i_0 \parallel i_1 \parallel \mathsf{nodes}[i])$$

where i_0, i_1 are such that $i = i_0 + 256 \cdot i_1$.

3.2.4 Seed trees

As explained in Section 2.4, the MQOM signature relies on seed trees. The latter are handled through the three following subrountines:

- TreePRG (Algorithm 4): it takes a 2λ-bit salt and a λ-bit seed, and returns N λ-bit seeds which correspond to the leaves of a binary seed tree with the given seed as root.
- GetSiblingPath (Algorithm 5): it takes a seed tree nodes, an index i^*, and it returns the sibling path of the seed leave indexed by i^* in the tree.
- GetSeedsFromPath (Algorithm 6): its takes an index i^*, a seed path, and a 2λ-bit salt, and it returns the seed leaves (except the one with index i^*) of the tree which would give this path when opening for i^*.

Algorithm 4 TreePRG

Input: A salt $\mathsf{salt} \in \{0,1\}^{2\lambda}$ and a seed $\mathsf{rseed} \in \{0,1\}^{\lambda}$
Output: $\mathsf{nodes}, \mathsf{seeds}$, a seed tree (with its $2^{D+1}-1$ nodes) and the 2^D seed leaves

1: $\mathsf{nodes}[1] = \mathsf{rseed}$
2: **for** i from 1 to $2^D - 1$ **do**
3: $\quad(\mathsf{nodes}[2i] \parallel \mathsf{nodes}[2i+1]) = \mathrm{Hash}_4(\mathsf{salt} \parallel i_0 \parallel i_1 \parallel \mathsf{nodes}[i])$ ▷ $i = i_0 + 256 \cdot i_1$
4: $\mathsf{seeds} = (\mathsf{nodes}[2^D], \ldots, \mathsf{nodes}[2^{D+1}-1])$
5: **return** $(\mathsf{nodes}, \mathsf{seeds})$ ▷ the seed tree, with its leaves

Algorithm 5 GetSiblingPath

Input: A seed tree nodes and an index $i^* \in \{1, \ldots, 2^D\}$
Output: The sibling path $\mathsf{path} \in \left(\{0,1\}^\lambda\right)^D$ of the seed leave indexed by i^*

$\mathsf{hidden\_node} = i^*$
$\mathsf{path} \leftarrow \emptyset$
for h from D to 1 **do**
    if hidden_node odd **then**
        $\mathsf{path} \leftarrow (\mathsf{path} \parallel \mathsf{nodes}[\mathsf{hidden\_node} - 1])$
    else
        $\mathsf{path} \leftarrow (\mathsf{path} \parallel \mathsf{nodes}[\mathsf{hidden\_node} + 1])$
    $\mathsf{hidden\_node} \leftarrow \left\lfloor \frac{\mathsf{hidden\_node}}{2} \right\rfloor$
return path

Algorithm 6 GetSeedsFromPath

Input: An index $i^* \in \{1, \ldots, 2^D\}$, the sibling path $\mathsf{path} \in \left(\{0,1\}^\lambda\right)^D$ of the seed leave indexed by i^*, and a salt $\mathsf{salt} \in \{0,1\}^{2\lambda}$
Output: The seed leaves of the corresponding trees, with $\mathsf{seeds}[i^*] = \texttt{null}$

$\mathsf{hidden\_node} = i^*$
for i from 1 to $2^{D+1} - 1$ **do**
    $\mathsf{nodes}[i] = \texttt{null}$
for h from D to 1 **do**
    $(\mathsf{seed} \parallel \mathsf{path}) \leftarrow \mathsf{path}$
    if hidden_node odd **then**
        $\mathsf{nodes}[\mathsf{hidden\_node} - 1] = \mathsf{seed}$
    else
        $\mathsf{nodes}[\mathsf{hidden\_node} + 1] = \mathsf{seed}$
    $\mathsf{hidden\_node} \leftarrow \left\lfloor \frac{\mathsf{hidden\_node}}{2} \right\rfloor$
for i from 1 to $2^D - 1$ **do**
    if $\mathsf{nodes}[i] \neq \texttt{null}$ **then**
        $(\mathsf{nodes}[2i] \parallel \mathsf{nodes}[2i+1]) = \mathrm{Hash}_4(\mathsf{salt} \parallel i_0 \parallel i_1 \parallel \mathsf{nodes}[i])$ ▷ $i = i_0 + 256 \cdot i_1$
$\mathsf{seeds} = (\mathsf{nodes}[2^D], \ldots, \mathsf{nodes}[2^{D+1} - 1])$
return seeds

3.3 Key generation

The key generation of MQOM consists in pseudorandomly generating an MQ instance, with triangular matrices A_i. It takes as input a root seed $\mathsf{seed}_{\mathsf{root}}$ from which it derives two further seeds $\mathsf{seed}_{\mathsf{x}}$ and $\mathsf{seed}_{\mathsf{eq}}$. The secret witness x is derived from $\mathsf{seed}_{\mathsf{x}}$ while the MQ equations $\{A_i, b_i\}$ are derived from $\mathsf{seed}_{\mathsf{eq}}$. The MQ output y is then computed from $\{A_i, b_i\}$ and x. Finally, the key pair is defined and returned as $pk := (\mathsf{seed}_{\mathsf{eq}}, y)$ and $sk := (\mathsf{seed}_{\mathsf{eq}}, y, x)$ with x and y in serialized form.

Algorithm 7 MQOM – Key Generation

$\mathsf{seed}_{\mathsf{root}} \leftarrow \{0,1\}^\lambda$
$(\mathsf{seed}_{\mathsf{x}}, \mathsf{seed}_{\mathsf{eq}}) \leftarrow \text{ExpandSeed}(\mathsf{salt} := 0, \mathsf{seed}_{\mathsf{root}}, 2)$ ▷ $\mathsf{seed}_{\mathsf{x}}, \mathsf{seed}_{\mathsf{eq}} \in \{0,1\}^\lambda$
$x \leftarrow \text{ExpandSecret}(\mathsf{seed}_{\mathsf{x}})$ ▷ $x \in \mathbb{F}_q^n$
$\{A_i, b_i\}_{i \in [1:m]} \leftarrow \text{ExpandMQEquations}(\mathsf{seed}_{\mathsf{eq}})$ ▷ $A_i \in \mathbb{F}_q^{n \times n}, b_i \in \mathbb{F}_q^n$
for $i \in [1:m]$ **do**
    $y_i = x^\intercal A_i x + b_i^\intercal x$ ▷ $y_i \in \mathbb{F}_q$
$\mathsf{x} = \text{Serialize}(x)$
$\mathsf{y} = \text{Serialize}(y)$
$pk = (\mathsf{seed}_{\mathsf{eq}}, \mathsf{y})$; $sk = (\mathsf{seed}_{\mathsf{eq}}, \mathsf{y}, \mathsf{x})$
return (pk, sk)

3.4 Signature algorithm

A detailed description of the signature algorithm is given in Algorithm 8 and Algorithm 9.

Algorithm 8 MQOM – Signature Algorithm – Part 1/2

Input: a secret key $sk = (\mathsf{seed_{eq}}, \mathsf{y}, \mathsf{x})$ and a message $m \in \{0,1\}^*$

▷ *Phase 0: Initialization.*

$\{A_i, b_i\}_{i\in[1:m]} \leftarrow \text{ExpandMQEquations}(\mathsf{seed_{eq}})$
$\mathsf{salt} \leftarrow \{0,1\}^{2\lambda}$
$\mathsf{mseed} \leftarrow \{0,1\}^{\lambda}$
$\{\mathsf{rseed}[e]\}_{e\in[1:\tau]} \leftarrow \text{ExpandSeed}(\mathsf{salt}, \mathsf{mseed}, \tau)$ ▷ $\mathsf{rseed}[e] \in \{0,1\}^{\lambda}$

▷ *Phase 1: Sharing and commitments.*

for $e \in [1:\tau]$ **do**
  $\big(\mathsf{tree}[e], \{\mathsf{seed}[e][i]\}_{i\in[1:N]}\big) \leftarrow \text{TreePRG}(\mathsf{salt}, \mathsf{rseed}[e])$
  $\mathsf{in\_mshare}[e][d] = 0$ for all $d \in [1:D]$ ▷ $\mathsf{in\_mshare}[e][d] \in \mathbb{F}_q^{n+\eta(n_2+2n_1-1)}$
  for $i \in [1:N]$ **do**
    XOF.Init($\mathsf{seed}[e][i]$)
    if $i \neq N$ **then**
      $\mathsf{in\_share}[e][i] \leftarrow \text{XOF.SampleFieldElements}(n + \eta(n_2 + 2n_1 - 1))$
      ▷ $\mathsf{in\_share}[e][i] \in \mathbb{F}_q^{n+\eta(n_2+2n_1-1)}$
      $\mathsf{com}[e][i] = \text{Commit}(\mathsf{salt}, e, i, \mathsf{seed}[e][i])$
      for $d \in [1:D]$: the d^{th} bit of $i-1$ is zero, **do**
        $\mathsf{in\_mshare}[e][d]$ += $\mathsf{in\_share}[e][i]$
    else
      $\mathsf{in\_acc}[e] = \sum_{i=1}^{N-1} \mathsf{in\_share}[e][i]$
      $(\mathsf{x\_acc}[e], \mathsf{unif\_acc}[e], \mathsf{hint\_acc}[e]) \leftarrow \text{Parse}(\mathsf{in\_acc}[e], \mathbb{F}_q^n, \mathbb{F}_q^{\eta\cdot n_2}, \mathbb{F}_q^{\eta(2n_1-1)})$
      $\mathsf{unif\_plain}[e] = \mathsf{unif\_acc}[e] + \text{XOF.SampleFieldElements}(n_2 \cdot \eta)$
      $\mathsf{x\_aux}[e] = \mathsf{x} - \mathsf{x\_acc}[e]$ ▷ $\mathsf{x\_aux}[e] \in \mathbb{F}_q^n$
      $\mathsf{com}[e][i] = \text{Commit}(\mathsf{salt}, e, i, \mathsf{seed}[e][i] \,\|\, \mathsf{x\_aux}[e])$

▷ *Phase 2: First challenge (random coefficients $\gamma_1, \ldots, \gamma_m$).*

$h_1 = \text{Hash}_1(m, \mathsf{salt}, \mathsf{com}[1][1], \ldots, \mathsf{com}[\tau][N])$
$\{\mathsf{chal}_1[e]\}_{e\in[1:\tau]} \leftarrow \text{ExpandFirstMPCChallenge}(h_1, \tau)$

▷ *Phase 3: Hint and commitments (polynomial Q').*

for $e \in [1:\tau]$ **do**
  $\mathsf{hint\_plain}[e] = \text{ComputePlainHint}(\mathsf{x}, \mathsf{unif\_plain}[e], \mathsf{chal}_1[e], (\{A_i, b_i\}_{i\in[1:m]}, \mathsf{y}))$
  $\mathsf{hint\_aux}[e] = \mathsf{hint\_plain}[e] - \mathsf{hint\_acc}[e]$ ▷ $\mathsf{hint\_aux}[e] \in \mathbb{F}_q^{\eta(2n_1)}$
  $\mathsf{com}'[e] = \text{Commit}(\mathsf{salt}, e, 0, \mathsf{hint\_aux}[e])$

Algorithm 9 MQOM – Signature Algorithm – Part 2/2

▷ *Phase 4: Second challenge (random evaluation point r).*

$h_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \mathsf{com}'[1], \ldots, \mathsf{com}'[\tau])$
$(\mathsf{chal}_2[e])_{e\in[1:\tau]} \leftarrow \mathrm{ExpandSecondMPCChallenge}(h_2, \tau)$

▷ *Phase 5: MPC simulation.*

for $e \in [1:\tau]$ **do**
  $\mathsf{in\_plain}[e] = (\mathsf{x}, \mathsf{unif\_plain}[e], \mathsf{hint\_plain}[e])$
  $\mathsf{broad\_plain}[e] \leftarrow \mathrm{ComputePlainBroadcast}(\mathsf{in\_plain}[e], \mathsf{chal}_1[e], \mathsf{chal}_2[e], \{A_i, b_i\}_{i\in[1:m]})$
  for $d \in [1:D]$ **do**
    $\mathsf{broad\_mshare}[e][d] = \mathrm{PartyComputation}\big(\mathsf{in\_mshare}[e][d], \mathsf{chal}_1[e], \mathsf{chal}_2[e], (\{A_i, b_i\}_{i\in[1:m]}, \mathsf{y}), \mathsf{broad\_plain}[e], \mathtt{False}\big)$
    ▷ $\mathsf{broad\_mshare}[e][d] \in \mathbb{F}_q^{(2d+1)t\eta}$

▷ *Phase 6: Third challenge (view opening).*

$h_3 = \mathrm{Hash}_3(m, \mathsf{salt}, h_2, \{\mathsf{broad\_plain}[e]\}_{e\in[1:\tau]}, \{\mathsf{broad\_mshare}[e][d]\}_{d\in[1:D],e\in[1:\tau]})$.
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \mathrm{ExpandViewChallenge}(h_3)$.

▷ *Phase 7: Views opening.*

for $e \in [1:\tau]$ **do**
  $\mathsf{path}[e] \leftarrow \mathrm{GetSiblingPath}(\mathsf{tree}[e], i^*[e])$.
  if $i^*[e] = N$ **then**
    $\mathsf{view}[e] = (\mathsf{path}[e], \mathsf{com}'[e])$
  else
    $\mathsf{view}[e] = (\mathsf{path}[e], \mathsf{x\_aux}[e], \mathsf{hint\_aux}[e])$

▷ *Phase 8: Signature.*

$\sigma = \big(\mathsf{salt} \parallel h_1 \parallel h_2 \parallel h_3 \parallel \{\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]]\}_{e\in[1:\tau]}\big)$
return σ

3.5 Verification algorithm

A detailed description of the verification algorithm is given in Algorithm 10 and Algorithm 11.

Algorithm 10 MQOM – Verification Algorithm – Part 1/2

Input: a public key $pk = (\mathsf{seed_{eq}}, \mathsf{y})$, a signature σ and a message $m \in \{0,1\}^*$

▷ *Phase 0: Parsing and expansion.*

$\big(\mathsf{salt} \parallel h_1 \parallel h_2 \parallel h_3 \parallel \{\mathsf{view}[e], \mathsf{broad\_plain}[e], \mathsf{com}[e][i^*[e]]\}_{e\in[1:\tau]}\big) = \sigma$
$\{A_i, b_i\}_{i\in[1:m]} \leftarrow \text{ExpandMQEquations}(\mathsf{seed_{eq}})$
$\mathsf{chal}_1 \leftarrow \text{ExpandFirstMPCChallenge}(h_1, \tau)$ ▷ Coefficients $\gamma_1, \ldots, \gamma_m$
$\mathsf{chal}_2 \leftarrow \text{ExpandSecondMPCChallenge}(h_2, \tau)$ ▷ Evaluation point r
$\{i^*[e]\}_{e\in[1:\tau]} \leftarrow \text{ExpandViewChallenge}(h_3)$ ▷ Unopened view indexes

for $e \in [1:\tau]$ **do** ▷ *Main loop (over the τ repetitions)*

▷ *(Main loop) Phase 0: Parsing views*

  if $i^*[e] = N$ **then**
    $(\mathsf{path}[e], \mathsf{com}'[e]) = \mathsf{view}[e]$
  else
    $(\mathsf{path}[e], \mathsf{x\_aux}[e], \mathsf{hint\_aux}[e]) = \mathsf{view}[e]$

▷ *(Main loop) Phase 1: Recomputing shares and commitments.*

  $(\mathsf{seed}[e][i])_{i\in[1:N\setminus i^*[e]]} \leftarrow \text{GetSeedsFromPath}(i^*[e], \mathsf{salt}, \mathsf{path}[e])$
  $\mathsf{in\_open\_mshare}[d] = 0$ for all $d \in [1:D]$ ▷ main party share **not** containing $i^*[e]$
  for $i \in [1:N]\setminus i^*[e]$ **do**
    XOF.Init($\mathsf{seed}[e][i]$)
    if $i \neq N$ **then**
      $\mathsf{in\_share}[e][i] \leftarrow \text{XOF.SampleFieldElements}(n + \eta(n_2 + 2n_1 - 1))$
      $\mathsf{com}[e][i] = \text{Commit}(\mathsf{salt}, e, i, \mathsf{seed}[e][i])$
    else
      $\mathsf{unif\_plain}[e][N] = \text{XOF.SampleFieldElements}(n_2 \cdot \eta)$
      $\mathsf{in\_share}[e][N] = (\mathsf{x\_aux}[e] \parallel \mathsf{unif\_plain}[e][N] \parallel \mathsf{hint\_aux}[e])$
      $\mathsf{com}[e][i] = \text{Commit}(\mathsf{salt}, e, i, (\mathsf{seed}[e][i] \parallel \mathsf{x\_aux}[e]))$
      $\mathsf{com}'[e] = \text{Commit}(\mathsf{salt}, e, 0, \mathsf{hint\_aux}[e])$
    for $d \in [1:D]$: the d^{th} bit of $i-1$ and $i^*[e]$ are different **do**
      $\mathsf{in\_open\_mshare}[e][d]$ += $\mathsf{in\_share}[e][i]$ ▷ $\mathsf{in\_open\_mshare}$ does not contain i^*

▷ *(Main loop) Phase 2: MPC simulation.*

  for $d \in [1:D]$ **do**
    if the d^{th} bit of $i^*[e]$ is 1 **then**
      $\mathsf{broad\_mshare}[e][d] =$
        $\text{PartyComputation}\big(\mathsf{in\_open\_mshare}[e][d], \mathsf{chal}_1[e], \mathsf{chal}_2[e], (\{A_i, b_i\}_{i\in[1:m]}, \mathsf{y}), \mathsf{broad\_plain}[e], \mathtt{False}\big)$
    else
      $\mathsf{broad\_mshare}[e][d] = (\mathsf{broad\_plain}[e] \parallel 0^\eta)$
        $- \text{PartyComputation}\big(\mathsf{in\_open\_mshare}[e][d], \mathsf{chal}_1[e], \mathsf{chal}_2[e], (\{A_i, b_i\}_{i\in[1:m]}, \mathsf{y}), \mathsf{broad\_plain}[e], \mathtt{True}\big)$

Algorithm 11 MQOM – Verification Algorithm – Part 2/2

▷ *Phase 3: Recomputing hashes.*

1: $h'_1 = \mathrm{Hash}_1(\mathsf{salt}, \mathsf{com}\,[1]\,[1], \ldots, \mathsf{com}\,[\tau]\,[N])$
2: $h'_2 = \mathrm{Hash}_2(m, \mathsf{salt}, h_1, \mathsf{com}'\,[1], \ldots, \mathsf{com}'\,[\tau])$
3: $h'_3 = \mathrm{Hash}_2(m, \mathsf{salt}, h_2, \{\mathsf{broad\_plain}\,[e]\}_{e\in[1:\tau]}, \{\mathsf{broad\_mshare}\,[e]\,[d]\}_{d\in[1:D],e\in[1:\tau]})$.

▷ *Phase 4: Verification.*

4: **return** $(h_1, h_2, h_3) \stackrel{?}{=} (h'_1, h'_2, h'_3)$

Remark 3. *In Step 28 of Algorithm 10,* $(\mathsf{broad\_plain}\,[e] \parallel 0^\eta)$ *means* $\mathsf{broad\_plain}\,[e]$ *padded with* η 0*'s which account for the expected plain value of* $v \in \mathbb{F}_{q^\eta}$*. The zero-valued* v *is omitted from the plain broadcast values to avoid increasing the size of the signature.*

4 Parameters and performances

In this section, we propose several parameter sets for the MQOM signature scheme. As explained hereafter, those parameters have been selected to meet the categories I, III and V defined by the NIST while targeting good performances (signature size and running times).

4.1 Selection of the parameters

MQ parameters. We first fixed the base field characteristic, testing values ranging from $q = 17$ to $q = 251$. For each tested q, we took the number of equations m to be equal the number of unknowns n and selected $m = n$ in order to achieve the target level of security (categories I, III and V) according to the *MQ estimator* [BMS+22] (see details in Section 5.2).

MPC parameters. At first we fix the number of parties to $N = 256$ (or equivalently the hypercube dimension $D = 8$) to achieve running times of few milliseconds while keeping short signatures. Then for each tested q, and given the selected MQ parameters $n = m$, we exhaust the relevant MPC parameters (n_1, n_2, η).

Given $(q, n, m, n_1, n_2, \eta)$ we deduce the number of repetitions τ necessary to achieve a forgery cost larger than λ bits, when λ is 128, 192 and 256 respectively for Categories I, III and V. Currently, the best forgery attack is obtained by applying the approach of [KZ20] and its cost is given by Equation 13 (see description Section 5.3). Then from $(q, n, m, n_1, n_2, \eta)$ and τ, we deduce the size of a signature. For each q (and associated $n = m$), we kept the MPC parameters (n_1, n_2, η) leading to the shortest signature. Table 3 summarizes the parameters and sizes we obtained for the different tested values of q.

Table 3: Tested fields $\mathbb{F}_q$, corresponding MQ parameters $n = m$, and optimal parameters for MQOM in terms of signature size (for $N = 256$).

q	$n = m$	n_1	n_2	η	τ	Size
17	54	5	11	10	20	6 528
19	53	5	11	10	20	6 528
23	51	4	13	10	20	6 489
29	50	5	10	10	20	6 368
31	49	5	10	10	20	6 348
$37 \to 53$	48	4	12	6	23	6 615
$59 \to 61$	47	4	12	6	23	6 615
$67 \to 73$	47	4	12	7	20	6 508
$79 \to 83$	46	4	12	7	20	6 488
$89 \to 127$	45	5	9	6	22	6 640
$131 \to 137$	45	5	9	5	22	6 618
$139 \to 173$	44	4	11	5	22	6 596
$179 \to 251$	43	4	11	5	22	6 575

We chose to propose the instances obtained for $q = 31$ and $q = 251$. The former achieves the shorter signature size for the tested fields. Moreover, an MQ instance with $q = 31$ was previously considered in the MQ-DSS [CHR+20] which might already have motivated some cryptanalysis

attempts on those parameters. On the other hand, $q = 251$ give the larger prime field whose elements hold in single bytes. Moreover it requires a smaller extension degree η than $q = 31$ which makes the underlying arithmetic faster. Such an instance might also be more amenable to future improvements of MPCitH based on threshold secret sharing [FR22].

Finally, we chose to add a "fast" variant relying on $N = 32$ parties. Compared to the "short" variant with $N = 256$, the "fast" variant is 2 to 3 times faster for the considered instances, for an overhead of $\sim$20% in the signature size.

4.2 Symmetric cryptography primitives

The MQOM signature scheme relies on two types of symmetric cryptography primitives: a hash function (Hash) which we instantiate with SHA3 [Dwo15], and an extendable output function (XOF) which we instantiate with SHAKE [Dwo15]. Table 4 summarizes the instances for each security category.

Table 4: Symmetric cryptography primitives for NIST Security Categories I, III, and V.

	Category I	Category III	Category V
Hash	SHA3-256	SHA3-384	SHA3-512
XOF	SHAKE-128	SHAKE-256	SHAKE-256

We recall here the usage of these symmetric primitives in the MQOM signature scheme (see Section 3 for details):

- Hash is used for
  - the commitments,
  - the Fiat-Shamir hashes h_1, h_2 and h_3,
  - the nodes of the seed trees.
- XOF is used for
  - the expansion of the MQ equations $\{A_i, b_i\}_i$,
  - the expansion of the hashes h_1, h_2 and h_3 into the MPC and view-opening challenges,
  - the expansion of the seeds (key generation and root seeds from master seed),
  - the expansion of the shares from the leaf seeds.

4.3 Polynomial interpolation

The MQOM signature scheme performs some polynomial interpolations as depicted by Equation 2 and Equation 6 (see the subroutines InterpolateX and InterpolateW). These interpolations rely on some distinct field elements $f_1, \ldots, f_{n_1} \in \mathbb{F}_q$. In our instances, we use

$$\begin{aligned} f_1 &:= 0, \\ f_2 &:= 1, \\ &\vdots \\ f_{n_1} &:= n_1 - 1. \end{aligned}$$

4.4 Extension fields

The MQOM signature scheme relies on an extension field $\mathbb{F}_{q^\eta}$ whose extension degree varies for the different variants (security category and/or short vs. fast variant). Table 5 summarizes the field extensions that we use in our instances.

Table 5: Definition of field extensions.

$\mathbb{F}_{q^\eta}$	Field extension
$\mathbb{F}_{31^6}$	$\mathbb{F}_{31}[u]/\langle u^6 - 3\rangle$
$\mathbb{F}_{31^7}$	$\mathbb{F}_{31}[u]/\langle u^7 - u - 3\rangle$
$\mathbb{F}_{31^8}$	$\mathbb{F}_{31}[u]/\langle u^8 - u^2 - 1\rangle$
$\mathbb{F}_{31^{10}}$	$\mathbb{F}_{31}[u]/\langle u^{10} - 3\rangle$
$\mathbb{F}_{31^{11}}$	$\mathbb{F}_{31}[u]/\langle u^{11} - u^3 - 1\rangle$
$\mathbb{F}_{251^2}$	$\mathbb{F}_{251}[u']/\langle u'^2 - 2\rangle$
$\mathbb{F}_{251^4}$	$\mathbb{F}_{251^2}[u]/\langle u^2 - (u' + 1)\rangle$
$\mathbb{F}_{251^5}$	$\mathbb{F}_{251}[u]/\langle u^5 - 3\rangle$
$\mathbb{F}_{251^7}$	$\mathbb{F}_{251}[u]/\langle u^7 - u - 1\rangle$

4.5 Keys and signature costs

Public key. The public key is of the format $pk := (\mathsf{seed}_{\mathsf{eq}}, \mathsf{y})$; consisting of a λ-bit seed $\mathsf{seed}_{\mathsf{eq}}$ representing the equations of the MQ instance, and a serialized vector $y \in \mathbb{F}_q^m$ corresponding to the outputs of the equations. For $q = 31$, we store 8 field elements on 5 bytes. For $q = 251$, we store one field element on one byte. The public key size is hence of

$$|pk| = \begin{cases} \frac{\lambda}{8} + \lceil \frac{5m}{8} \rceil \text{ bytes} & \text{for } q = 31 \\ \frac{\lambda}{8} + m \text{ bytes} & \text{for } q = 251 \end{cases}$$

Secret key. The secret key is of the format $sk := (\mathsf{seed}_{\mathsf{eq}}, \mathsf{y}, \mathsf{x})$; consisting of the same elements as the public key, with the witness x which is a serialized vector $x \in \mathbb{F}_q^n$. Thus, the size of the secret key is

$$|sk| = \begin{cases} \frac{\lambda}{8} + \lceil \frac{5m}{8} \rceil + \lceil \frac{5n}{8} \rceil \text{ bytes} & \text{for } q = 31 \\ \frac{\lambda}{8} + m + n \text{ bytes} & \text{for } q = 251 \end{cases}$$

As all the existing public-key schemes, let us remark that we have an alternative definition of the key generation in which the secret key would be $\mathsf{seed}_{\mathsf{root}}$, the seed from which $(\mathsf{seed}_{\mathsf{eq}}, \mathsf{y}, \mathsf{x})$ are derived. In that case, the size of the secret key would be of $\lambda/8$ bytes, but the signer would need to recompute $\mathsf{seed}_{\mathsf{eq}}$, y and x at each signature, increasing the running time of the signing process. Moreover, the signature algorithm would be more sensitive to side-channel attacks. We recommend to use this alternative *only when the size of the secret key is critical*.

Signature size. The theoretical size (in bits) of a signature is:

$$
\begin{aligned}
\text{Total Size} &= 2\lambda && \text{size of the } \mathsf{salt}.\\
&+ 6\lambda && \text{size of } h_1,\ h_2 \text{ and } h_3.\\
&+ \tau \cdot (n \cdot \log_2(q)) && \text{size of } \mathsf{aux\_x}[e] \text{ in } \mathsf{view}[e].\\
&+ \tau \cdot ((2n_1 - 1) \cdot \eta \cdot \log_2(q)) && \text{size of } \mathsf{aux\_hint}[e] \text{ in } \mathsf{view}[e].\\
&+ \tau \cdot (n_2 \cdot \eta \cdot \log_2(q)) && \text{size of } \mathsf{plain\_br}[e].\\
&+ \tau \cdot \lambda \cdot \log_2(N) && \text{size of } \mathsf{path}[e] \text{ in } \mathsf{view}[e].\\
&+ \tau \cdot 2\lambda && \text{size of } \mathsf{com}[e][i^*[e]].\\
&+ \tau \cdot 2\lambda && \text{size of } \mathsf{com}'[e] \text{ in } \mathsf{view}[e].
\end{aligned}
$$

Given our encoding on field elements, $\log_2(q)$ should be replaced by 5 for $q = 31$ and by 8 for $q = 251$. We obtain the following approximate sizes in bytes:

$$|\sigma_{q=31}| = \lambda + \frac{5\tau}{8} \cdot \big(n + \eta \cdot (2n_1 + n_2 - 1)\big) + \frac{\tau \cdot \lambda \cdot (\log_2 N + 4)}{8}$$

and

$$|\sigma_{q=251}| = \lambda + \tau \cdot \big(n + \eta \cdot (2n_1 + n_2 - 1)\big) + \frac{\tau \cdot \lambda \cdot (\log_2 N + 4)}{8}$$

4.6 Proposed instances

All the signature parameters are summarized in Table 6 and in Table 7. Table 6 gives the parameters which are common to both variants while Table 7 gives the additional parameters.

Table 6: The MQ and MPC parameters of MQOM for NIST Security Categories I, III, and V.

Parameter Sets	**NIST Security**		**MQ Parameters**		**MPC Parameters**				
	Category	Bits	q	$m = n$	$N = 2^D$	n_1	n_2	η	τ
MQOM-L1-gf31-short	I	143	31	49	256	5	10	10	20
MQOM-L1-gf31-fast	I	143	31	49	32	5	10	6	35
MQOM-L1-gf251-short	I	143	251	43	256	4	11	5	22
MQOM-L1-gf251-fast	I	143	251	43	32	4	11	4	34
MQOM-L3-gf31-short	III	207	31	77	256	6	13	11	30
MQOM-L3-gf31-fast	III	207	31	77	32	6	13	7	51
MQOM-L3-gf251-short	III	207	251	68	256	5	14	7	30
MQOM-L3-gf251-fast	III	207	251	68	32	5	14	4	52
MQOM-L5-gf31-short	V	272	31	106	256	6	18	10	42
MQOM-L5-gf31-fast	V	272	31	106	32	6	18	8	66
MQOM-L5-gf251-short	V	272	251	93	256	6	16	7	41
MQOM-L5-gf251-fast	V	272	251	93	32	6	16	5	66

Table 7: The key and signature sizes in bytes.

Parameter Set	**Sizes (in bytes)**			
	pk	sk	Sig. Avg	Sig. Max
MQOM-L1-gf31-short	47	78	6 348	6 352
MQOM-L1-gf251-short	59	102	6 575	6 578
MQOM-L1-gf31-fast	47	78	7 621	7 657
MQOM-L1-gf251-fast	59	102	7 809	7 850
MQOM-L3-gf31-short	73	122	13 837	13 846
MQOM-L3-gf251-short	92	160	14 257	14 266
MQOM-L3-gf31-fast	73	122	16 590	16 669
MQOM-L3-gf251-fast	92	160	17 161	17 252
MQOM-L5-gf31-short	99	166	24 147	24 158
MQOM-L5-gf251-short	125	218	24 926	24 942
MQOM-L5-gf31-fast	99	166	28 917	29 036
MQOM-L5-gf251-fast	125	218	29 919	30 092

4.7 Benchmarks

Benchmarks for

- a reference implementation on an AVX2 machine are given in Table 8;
- an optimized implementation on an AVX2 machine are given in Table 9.

Table 8: Benchmark of **reference implementation** of the MQOM on an AVX2 machine. Timings were run on an Intel i7 at 3.8GHz while disabling Intel Turbo Boost.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM-L1-gf251-fast	0.15	0.57M	3.80	14.42M	3.43	13.02M
MQOM-L3-gf251-fast	0.61	2.33M	10.17	38.58M	9.23	34.99M
MQOM-L5-gf251-fast	1.51	5.74M	24.48	92.82M	22.76	86.30M
MQOM-L1-gf31-fast	0.19	0.73M	5.42	20.54M	4.80	18.21M
MQOM-L3-gf31-fast	0.74	2.80M	16.26	61.67M	14.92	56.57M
MQOM-L5-gf31-fast	1.93	7.30M	43.85	166.29M	41.16	156.10M
MQOM-L1-gf251-short	0.17	0.65M	12.83	48.64M	12.37	46.91M
MQOM-L3-gf251-short	0.63	2.40M	25.71	97.50M	24.53	93.03M
MQOM-L5-gf251-short	1.52	5.75M	55.68	211.15M	53.78	203.95M
MQOM-L1-gf31-short	0.20	0.74M	14.67	55.63M	13.86	52.57M
MQOM-L3-gf31-short	0.75	2.86M	34.71	131.62M	32.96	124.99M
MQOM-L5-gf31-short	1.92	7.28M	74.07	280.87M	71.04	269.38M

Table 9: Benchmark of **optimized implementation** of the MQOM on an AVX2 machine. Timings were run on an Intel i7 at 3.8GHz while disabling Intel Turbo Boost.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM-L1-gf251-fast	0.13	0.48M	3.04	11.52M	2.68	10.16M
MQOM-L3-gf251-fast	0.52	1.96M	8.66	32.85M	7.80	29.58M
MQOM-L5-gf251-fast	1.27	4.83M	21.51	81.55M	19.93	75.58M
MQOM-L1-gf31-fast	0.17	0.65M	4.66	17.65M	4.10	15.54M
MQOM-L3-gf31-fast	0.65	2.46M	14.84	56.29M	13.51	51.25M
MQOM-L5-gf31-fast	1.68	6.38M	41.21	156.26M	38.56	146.20M
MQOM-L1-gf251-short	0.13	0.48M	7.52	28.51M	7.20	27.31M
MQOM-L3-gf251-short	0.52	1.98M	18.33	69.51M	17.29	65.56M
MQOM-L5-gf251-short	1.28	4.86M	39.03	148.00M	37.51	142.26M
MQOM-L1-gf31-short	0.18	0.67M	11.70	44.36M	11.00	41.72M
MQOM-L3-gf31-short	0.66	2.51M	28.52	108.13M	26.96	102.22M
MQOM-L5-gf31-short	1.65	6.24M	59.19	224.45M	56.33	213.61M

Benchmark Platform. Intel(R) Core(TM) i7-10700K CPU 3.80GHz. The scheme has been compiled with Clang compiler (`Apple clang version 14.0.0`) with options

`-O3 -flto -march=native`.

5 Security analysis

5.1 Proof of unforgeability

The MQOM signature scheme aims at providing *unforgeability against chosen message attacks* (EUF-CMA). In this setting, the adversary is given a public key pk and they can ask an oracle (called the *signature oracle*) to sign messages $(m_1, \ldots, m_r)$ that they can select at will. The goal of the adversary is to generate a pair (m, σ) such that m is not one of requests to the signature oracle and such that σ is a valid signature of m with respect to pk.

Our security statement is based on the following assumptions:

- **MQ hardness.** Solving the considered MQ instance is $(\epsilon_{\text{MQ}}, t)$-*hard* for some $(\epsilon_{\text{MQ}}, t)$ which are implicit functions of the security parameter λ. Formally, any adversary $\mathcal{A}$ on input a random MQ instance $\big(\{A_i\}, \{b_i\}, y\big)$ and running in time at most t has probability at most ϵ_{MQ} to output the solution x of the input instance.
- **Secure pseudorandmness.** The pseudorandmness generated by the extendable output hash function (XOF) is indistinguishable from true randomness (provided that the input seed has sufficient entropy). Formally, an adversary $\mathcal{A}$ running in time at most t has advantage at most ϵ_{PRG} in distinguishing the two following distributions:

  $$\mathcal{D}_0 = \{x \leftarrow \{0,1\}^{\ell_{\max}}\} \quad \text{and} \quad \mathcal{D}_0 = \{x \in \{0,1\}^{\ell_{\max}} \leftarrow \text{XOF}(\mathsf{seed}) \mid \mathsf{seed} \leftarrow \mathcal{S}\}$$

  for any distribution $\mathcal{S}$ with at least λ bits of min-entropy and where $\ell_{\max}$ denotes the maximum number of bits sampled from XOF with overwhelming probability (*i.e.* ignoring negligible occurrence of long sequences due to rejection sampling).
- **ROM.** The hash functions Hash_0, Hash_1, Hash_2, Hash_3, Hash_4 behaves as random oracles. Formally, our security statement only holds in the random oracle model where the Hash_i's are modelled as random oracles.

Under the above security assumptions, we get the following result:

Theorem 5.1. *Let* Hash_0, Hash_1, Hash_2, Hash_3, Hash_4 *be modelled as random oracles, and let* p_1, p_2 *be the probabilities from Theorem 2.1. Let* $\mathcal{A}$ *be an adversary against the EUF-CMA security of the scheme running in time* t *and making a total of* q_i *queries to* Hash_i *for* $i \in \{1,2,3,4\}$ *and* q_{sign} *queries to the signing oracle. Under the above assumptions,* $\mathcal{A}$*'s advantage in the EUF-CMA game is upper-bounded as*

$$\epsilon_{\text{EUF-CMA}} \leq \epsilon_{\text{MQ}} + \epsilon_{\text{PRG}} + \frac{cst \cdot (q_1 + q_2 + q_3 + q_4 + q_{sign})^2}{2^{2\lambda}} + \Pr[X + Y + Z = \tau],$$

for some constant cst *(which depends on the parameters* N *and* τ*) and with*

- $X = \max_{i \in [1:q_1]}\{X_i\}$ *with* $X_i \sim \mathcal{B}(\tau,\, p_1)$,
- $Y = \max_{i \in [1:q_2]}\{Y_i\}$ *with* $Y_i \sim \mathcal{B}(\tau - X,\, p_2)$,
- $Z = \max_{i \in [1:q_3]}\{Z_i\}$ *with* $Z_i \sim \mathcal{B}\big(\tau - X - Y,\, \frac{1}{N}\big)$,

where $\mathcal{B}(n_0, p_0)$ *denotes the binomial distribution with* n_0 *the number of trials and* p_0 *the success probability of each trial.*

The proof of the above theorem is very similar to, and can be easily adapted from, the security proof of the Banquet signature scheme [BDK+21].

5.2 Attacks against MQ instances

The security of the MQOM signature scheme relies on the hardness to solve an instance of the multivariate quadratic problem, since the secret key is a solution of the MQ instance represented by the public key. There exists many algorithms to solve the MQ problem. Their complexity depends on several parameters: the number n of unknowns, the number m of quadratic equations, the size q of the field, the characteristic of the field, and the number of solutions. The optimal algorithm might vary depending on the values of these parameters.

We used the MQ estimator [BMS+22] to evaluate the computational costs of all the existing algorithms. This tool takes the parameters (q, m, n) of an MQ problem and estimates the running times (and the memory usage) of the most important classical algorithms. The current best known algorithms all combine algebraic techniques with exhaustive search. We focus our analysis on the three following algorithms:

- Hybrid-(F4/F5) [Fau99; Fau02]: the idea is to guess a set of k variables and then to solve the resulting MQ system by running the F5 algorithm. The latter is an algorithm to compute a Gröbner basis of an ideal of polynomial equations.
- FXL [CKP+00; BFS+13]: it consists in guessing a set of k variables and to check the consistency of the resulting problem thanks to a Macaulay matrix at large enough degree.
- Crossbred [JV18]: the idea is to first perform some operations on the Macaulay matrix of the given system, and only afterward to fix k variables.

The MQ estimator [BMS+22] takes two parameters as inputs:

- a real number $w \in [2, 3]$ such that the complexity to multiply two $n \times n$ matrices is $O(n^w)$.
- a real number $\theta \in [0, 2]$ such that $(\log_2 q)^\theta$ is the ratio between the field operation complexity and the bit complexity.

Regarding the complexity of the matrix multiplication, we set w as $\log_2(7)$. The same choice was made in MQ-DSS [CHR+20] and in the numerical results of the MQ estimator [BMS+22]. While there exist some algorithms with asymptotically smaller w, those algorithms have huge constant factor. In practice, the best an adversary can hope is $w = \log_2(7)$ obtained using Strassen algorithm [Str69]. Moreover, we take $\theta = 2$ as suggested in [BMS+22].

We only consider the MQ instances for which the number n of unknowns and the number m of equations are the same (*i.e.* $m = n$), since they corresponds to the harder instances. Indeed, adding more equations would provide information about the system, while having more variables would enales us to reduce the instance by fixing some of them. Table 10 provides the MQ estimator output for our selected MQ parameters for the three security categories.

Table 10: Complexities of different attacks on our MQ instances.

Parameter Set	MQ Parameters		Algorithms		
	q	$m = n$	HybridF5	FXL	Crossbred
MQOM-L1-gf31	31	49	164.2	142.8	143.9
MQOM-L1-gf251	251	43	168.9	144.4	151.6
MQOM-L3-gf31	31	77	244.6	207.6	232.6
MQOM-L3-gf251	251	68	251.4	209.8	262.1
MQOM-L5-gf31	31	106	326.9	276.2	325.8
MQOM-L5-gf251	251	93	334.9	274.1	369.3

5.3 Signature forgery attacks

While applying the Fiat-Shamir transform to a zero-knowledge proof of knowledge (ZK-PoK) with several rounds and parallel repetitions, one gets a security drop between the soundness of the ZK-PoK and the unforgeability of the signature scheme. Namely, the forgery cost is lower than $\frac{1}{\varepsilon}$, where ε is the soundness error of the original ZK-PoK. The best forgery attack against Fiat-Shamir-based schemes with several parallel executions ($\tau > 1$) is the attack from [KZ20]. The latter is described for a 5-round scheme, but it can be easily generalized for any scheme with more rounds. The attack works as follows:

1. The adversary chooses an attack strategy. It consists to choose three non-negative integers τ_1, τ_2, τ_3 such that $\tau = \tau_1 + \tau_2 + \tau_3$.

2. For each forgery attempt, they generate cheating commitments (*i.e.* which do not correspond to a correct witness) which for each repetition fool the MPC protocol with probability p_1 in round 1 (commitment – challenge 1) and probability p_2 in round 2 (response – challenge 2), where p_1, p_2 are as defined in Theorem 2.1. They repeat this step until the MPC protocol is fooled in round 1 for τ_1 repetitions among τ. Since the probability that a cheating commitment fools the MPC protocol w.r.t. a random first challenge (expanded from h_1) is p_1 for one repetition, the probability to get τ_1 fooled repetitions is

$$\mathsf{PMF}(\tau, \tau_1, p_1) := \Pr\left(\begin{array}{c}\text{fool at least } \tau_1 \text{ of the } \tau \\ \text{repetitions in round 1 with} \\ \text{unitary probability of } p_1\end{array}\right) = \sum_{k=\tau_1}^{\tau} \binom{\tau}{k} p_1^k (1-p_1)^{\tau-k} \ .$$

   Thus this step is repeated $\frac{1}{\mathsf{PMF}(\tau,\tau_1,p_1)}$ times in average. We denote $I_1 \subseteq [1:\tau]$ the set of indices of the fooled repetitions for a forgery attempt successfully passing this step.

3. For a forgery attempt successfully passing the previous step, the adversary tries different round-2 responses to the first challenge generated from h_1. Each response attempt gives rise to a different hash h_2 and corresponding random round-2 challenge which can be fooled with probability p_2 for each repetition. This step is repeated until the generated challenge is fooled for τ_2 repetitions among $[1:\tau] \setminus I_1$. The probability to succeed for one round-2 response attempt is

$$\mathsf{PMF}(\tau - \tau_1, \tau_2, p_2) = \sum_{k=\tau_2}^{\tau-\tau_1} \binom{\tau-\tau_1}{k} p_2^k (1-p_2)^{\tau-\tau_1-k} \ .$$

Thus this step is repeated $\frac{1}{\mathsf{PMF}(\tau-\tau_1,\tau_2,p_2)}$ times in average. We denote $I_2 \subseteq [1:\tau] \setminus I_1$ the set of indices of the fooled repetitions for a forgery attempt successfully passing this step.

4. Finally, the adversary guesses the unopened parties' indexes in round 3 for the remaining repetitions $e \in [1:\tau]\setminus(I_1 \cup I_2)$ (the repetitions which have not been fooled yet). They generate the round-3 responses accordingly, *i.e.* cheating on the guessed non-opened party only and such that the broadcast shares pass the verification (*i.e.* correspond to $v = 0$).They repeat this step until the guess is correct, which happens with probability

$$\left(\frac{1}{N}\right)^{\tau-\tau_1-\tau_2}.$$

Thus this last step is repeated $N^{\tau-\tau_1-\tau_2}$ times in average.

The forgery cost of this attack corresponds to the cost of the optimal strategy, namely

$$\mathsf{cost}_{\mathsf{forge}} = \max_{\tau_1,\tau_2,\tau_3:\tau=\tau_1+\tau_2+\tau_3} \left\{ \frac{1}{\mathsf{PMF}(\tau,\tau_1,p_1)} + \frac{1}{\mathsf{PMF}(\tau-\tau_1,\tau_2,p_2)} + N^{\tau_3} \right\}, \tag{13}$$

where p_1 and p_2 are defined in Theorem 2.1. In practice, the parameters in Table 6 have been chosen such that the associated forgery cost is at least of 2^{128} for Category I, of 2^{192} for Category III and of 2^{256} for Category V. We stress that here a forgery cost of 2^λ means computing more than 2^λ hashes, which is relevant with respect to the definition of Categories I, II and V (assuming that a hash computation is comparable or superior to an AES computation in gate count).

6 Advantages and limitations

Bad news first, the MQOM signature scheme suffers the following limitations:

- **Relatively slow:** As other MPCitH based scheme, MQOM is relatively slow, with signing and verification time ranging between $10 \cdot 10^6$ and $45 \cdot 10^6$ cycles (2.7 – 11.7 ms on Intel i7 3.8 GHz processor) for NIST security category 1. This is slow compared to lattice-based signatures. One of the reason is the greedy use of symmetric cryptography, notably (output extendable) hashing, which let room from drastic improvement on platform supporting hashing in hardware.
- **Relatively large signatures:** Compared to lattice-based signature or post-quantum signatures based on the hash-and-sign paradigm, MPCitH-based signatures such as MQOM have larger sizes. But in comparison to several hash-and-sign schemes, things are different if we further count the size of the public key (as stressed hereafter).
- **Quadratic growth in the security level:** As other MPCitH-based signature schemes, or, more generally, as other schemes applying the Fiat-Shamir transform to a parallelly repeated ZK-PoK with non-negligible soundness error, MQOM suffers a quadratic growth of the signature size. In practice, the size of MQOM signatures roughly doubles while going from Category I to Category III and while going from Category III to Category V as well.

On the other hand, MQOM benefits of the following advantages:

- **Conservative hardness assumption:** Being generic, the MPCitH approach can be applied to any problem on does not rely on structured problems to introduce a trapdoor. MQOM benefits this by relying on a full random instance of the MQ problem which is believe to be a conservative hardness assumption.
- **Small (public) keys:** Thanks to the unstructured feature of the MQ instance, it can be mostly derive from a random seed. Hence the public key is only composed of a λ-bit seed and the relatively-short output y of the MQ system. The secret key additionally includes the relatively-short input x of the MQ system (which can further be fully compressed as the root seed of the key generation).
- **Highly parallelizable:** As other schemes based on the MPCitH paradigm, MQOM is highly parallelizable. Most of the computation can be done in parallel for the τ repetitions and computation can be further parallelized inside a repetition ($D(+1)$ party computations, seed trees and commitments).
- **Good public key + signature size:** As other schemes based on the MPCitH paradigm, MQOM achieves a good score in terms of "public key + signature size" metric compared to other candidate post-quantum signature schemes which are not based on lattices.
- **Relatively small signatures:** MPCitH-based signature schemes in the literature (at time of writing) have signature sizes ranging on 5–10 KB (for 128-bit of security). MQOM is on the lower side of this range, with 6.3–6.6 KB. Moreover, MPCitH-based signatures achieving lower sizes are based on arguably less conservative assumptions such as *e.g.* recent dedicated symmetric designs or rank metric problems (MinRank, Rank Syndrome Decoding).

References

[AGH+23] C. Aguilar Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The return of the SDitH. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part V*, pages 564–596. Springer, Heidelberg, 2023 (cited on pages 1, 10, 12).

[BDK+21] C. Baum, C. Delpech de Saint Guilhem, D. Kales, E. Orsini, P. Scholl, and G. Zaverucha. Banquet: short and fast signatures from AES. In J. Garay, editor, *PKC 2021, Part I*, pages 266–297. Springer, Heidelberg, 2021 (cited on pages 1, 4, 37).

[BFS+13] M. Bardet, J.-C. Faugère, B. Salvy, and P.-J. Spaenlehauer. On the complexity of solving quadratic boolean systems. *Journal of Complexity*, (1):53–75, 2013 (cited on page 38).

[BMS+22] E. Bellini, R. H. Makarim, C. Sanna, and J. A. Verbel. An estimator for the hardness of the MQ problem. In L. Batina and J. Daemen, editors, *AFRICACRYPT 22*, pages 323–347. Springer Nature, 2022 (cited on pages 31, 38).

[CHR+20] M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe. MQDSS specifications. Version 2.1, 2020. https://mqdss.org/files/mqdssVer2point1.pdf (cited on pages 31, 38).

[CKP+00] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preneel, editor, *EUROCRYPT 2000*, pages 392–407. Springer, Heidelberg, 2000 (cited on page 38).

[DOT21] C. Delpech de Saint Guilhem, E. Orsini, and T. Tanguy. Limbo: efficient zero-knowledge MPCitH-based arguments. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 3022–3036. ACM Press, 2021 (cited on pages 1, 4).

[Dwo15] M. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (cited on page 32).

[Fau02] J. C. Faugère. A new efficient algorithm for computing gröbner bases without reduction to zero (f5). In *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, 75–83, Lille, France. Association for Computing Machinery, 2002. ISBN: 1581134843 (cited on page 38).

[Fau99] J.-C. Faugére. A new efficient algorithm for computing gröbner bases (f4). *Journal of Pure and Applied Algebra*, (1):61–88, 1999 (cited on page 38).

[Fen22] T. Feneuil. Building MPCitH-based signatures from MQ, MinRank, rank SD and PKP. Cryptology ePrint Archive, Report 2022/1512, 2022. https://eprint.iacr.org/2022/1512 (cited on pages 1, 4).

[FR22] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. Cryptology ePrint Archive, Report 2022/1407, 2022. https://eprint.iacr.org/2022/1407 (cited on pages 3, 32).

[IKO+07] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In D. S. Johnson and U. Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, 2007 (cited on page 1).

[JV18] A. Joux and V. Vitse. A crossbred algorithm for solving boolean polynomial systems. In J. Kaczorowski, J. Pieprzyk, and J. Pomykała, editors, *Number-Theoretic Methods in Cryptology*, pages 3–21, Cham. Springer International Publishing, 2018 (cited on page 38).

[KKW18] J. Katz, V. Kolesnikov, and X. Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, 2018 (cited on pages 1, 10, 11).

[KZ20] D. Kales and G. Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In S. Krenn, H. Shulman, and S. Vaudenay, editors, *CANS 20*, pages 3–22. Springer, Heidelberg, 2020 (cited on pages 31, 39).

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, (13), 1969. https://doi.org/10.1007/BF02165411 (cited on page 38).