MQOM: MQ on my Mind

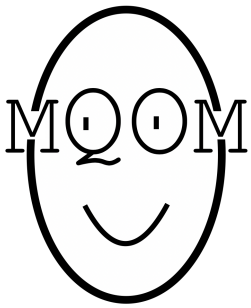
Algorithm Specifications and Supporting Documentation (Version 2.0)

Ryad Benadjila Charles Bouillaguet Thibauld Feneuil

Matthieu Rivain

February 5, 2025

CryptoExperts, Sorbonne Université



Contents

1 Introduction

MQOM (MQ-On-my-Mind) is a signature scheme derived from a zero-knowledge proof-of-knowledge of a secret solution to a random MQ instance. This zero-knowledge proof leverages the MPC-in-the-Head (MPCitH) paradigm [IKO+07] and is converted into a signature scheme using the Fiat-Shamir heuristic. This document specifies MQOM v2, the second version of the MQOM signature scheme, a second round candidate to the NIST *call for additional digital signature schemes for the post-quantum cryptography standardization process* [NIS22].

The proof system in MQOM v2 builds upon the Threshold-Computation-in-the-Head (TCitH) framework [FR23b]. Like the proof system in MQOM v1 [FR23a; BFR24], this framework transforms an MPC protocol into a zero-knowledge proof-of-knowledge via the MPCitH paradigm, while committing to secret sharings using GGM seed trees (as first proposed in [KKW18]). However, the TCitH framework utilizes threshold (Shamir) secret sharing instead of additive secret sharing. This shift reduces the computational cost of MPC emulation and enables the exploitation of multiplication homomorphism, offering significant performance improvements. The TCitH proof system in MQOM v2 can also be interpreted as a variant of VOLE-in-the-Head [BBD+23], where small VOLE correlations are directly applied in parallel repetitions, rather than being combined into a larger VOLE correlation.

By transitioning from the original MPCitH proof system (relying on additive secret sharing) to the TCitH proof system, the size of MQOM signatures has been roughly halved. In particular, for Category I, MQOM v2 achieves signatures of 2.8–4.2 KB against 6.3–7.9 KB for MQOM v1. Unless otherwise specified, MQOM should refer to MQOM v2 in the rest of this documentation.

Organization of the document. Section 2 gives an overview of the MQOM signature scheme as well as a detailed description of the key generation, signature and verification algorithms and their underlying subroutines. Section 3 explains the selection of parameters and depicts the proposed instances and their performances. Section 4 provides a security analysis of the MQOM signature scheme. Section 5 discusses the design choices of MQOM, while Section 6 addresses its advantages and limitations.

2 Description of the MQOM signature scheme

2.1 Overview

The Threshold-Computation-in-the-Head (TCitH) framework specializes the MPC-in-the-Head paradigm with threshold (Shamir) secret sharing [FR23c; FR23b]. In this framework, the prover commits to a Shamir secret sharing $[\![x]\!]$ of the secret value x and simulates an MPC protocol to verify the validity of x (e.g., as the solution to a public MQ instance). During this process, the prover reveals the publicly broadcast sharings $[\![\alpha]\!]$ to the verifier. The verifier, in turn, checks certain properties of $[\![\alpha]\!]$ to assess the validity of x and finally challenge the prover to open specific parties to verify the correctness of the MPC simulation. The TCitH framework is closely related to the VOLE-in-the-Head (VOLEitH) framework [BBD+23], where the prover commits to VOLE correlations of the form $x \cdot \Delta + r_x$ (for a random r_x). The prover then executes a protocol that computes and transmits to the verifier VOLE correlations of the form $\alpha \cdot \Delta + r_\alpha$ (analogous to the broadcast sharings $[\![\alpha]\!]$), before ultimately revealing $x \cdot \Delta + r_x$ for a challenge value of Δ.

Both frameworks can be interpreted as composing of a *polynomial interactive oracle proof* (polynomial IOP or PIOP) and a (zero-knowledge) *polynomial commitment scheme* (PCS), an increasingly common approach in the design of proof systems [SZ22; Tha23]. In the case of TCitH, committing to a Shamir secret sharing $[\![x]\!]$ corresponds to committing to the underlying polynomial P_x. Revealing a share $[\![x]\!]_i$ is equivalent to disclosing an evaluation $P_x(\omega_i)$. Similarly, VOLEitH commits to a VOLE correlation, which corresponds to a degree-1 polynomial $P_x(\Delta) = x \cdot \Delta + r_x$, with the prover later revealing an evaluation of this polynomial.

This section provides an overview of the MQOM signature scheme and the underlying TCitH-Π_{PC} proof system [FR23b] within the PIOP+PCS formalism. We begin by recalling the definition of the MQ problem. Next, we introduce the PIOP and PCS components that define the MQOM zero-knowledge proof of knowledge (ZK PoK). Finally, we discuss the compilation of this ZK PoK into the MQOM signature scheme using parallel repetitions, the Fiat-Shamir transform, and additional optimizations.

2.1.1 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 $\mathbb{F}$ be a finite field and let m, n be positive integers. The Multivariate Quadratic (MQ) problem with parameters $(\mathbb{F}, m, n)$ is the following problem:*

Let $(A_i)_{i\in[m]}$, $(b_i)_{i\in[m]}$, x and $y = (y_1, \ldots, y_m)$ be such that:

1. *x is uniformly sampled from $\mathbb{F}^n$,*
2. *for all $i \in [m]$, A_i is uniformly sampled from $\mathbb{F}^{n\times n}$,*
3. *for all $i \in [m]$, b_i is uniformly sampled from $\mathbb{F}^n$,*
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.

2.1.2 MQOM polynomial IOP

Formally, a PIOP is an interactive proof in which the prover can send a *polynomial oracle* $[P_1, \ldots, P_n]$ to the verifier for polynomials $P_1, \ldots, P_n \in \mathbb{K}[X]$ of prescribed degree d. From such a polynomial oracle, the verifier can then query some evaluations. Namely, for a query r to the oracle, the latter provides the verifier with the polynomial evaluations $P_1(r), \ldots, P_n(r)$. The verifier has the guaranty that some polynomial of degree d are embedded in the oracle and that their evaluations in r match the oracle's response.

In the MQOM PIOP, the prover aims to convince the verifier that they know an MQ solution $x \in \mathbb{F}^n$ such that $F(x) = (0, \ldots, 0) \in \mathbb{F}^m$, where

$$F = (f_1, \ldots, f_m) \quad \text{with} \quad f_i : x \mapsto x^\intercal A_i x + b_i^\intercal x - y_i \ . \tag{1}$$

This protocol is the PIOP equivalent of the QuickSilver protocol [YSW+21] within the VOLE-in-the-Head framework [BBD+23] or the Π_{PC} MPC protocol within the TCitH framework [FR23b].

Notions. Let $\mathbb{K}$ be an extension field of $\mathbb{F}$ and $\Omega \subseteq \mathbb{K}$, an *evaluation domain* (i.e. the points on which the polynomial oracle can be queried). We denote $\mathbb{K}[X]^{\leqslant d}$ the set of polynomials of degree $\leq d$ with coefficients in $\mathbb{K}$ and we shall consider *vector polynomials*, which are vectors with polynomial coordinates. We consider the homogeneous application of F to a degree-1 vector polynomial $P \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^n$, which results in a degree-2 vector polynomial $F(P) \in \left(\mathbb{K}[X]^{\leqslant 2}\right)^m$ such that

$$F(P)(X) = P(X)^\intercal A_i P(X) + b_i^\intercal P(X) \cdot X - y_i \cdot X^2 \ .$$

We shall denote by $P(\infty) \in \mathbb{K}^n$ the leading coefficient vector of a vector polynomial P. In particular, for $P \in \mathbb{K}[X]^{\leqslant 1}$, $P(\infty)$ denotes the degree-1 coefficient vector, while for $F(P) \in \mathbb{K}[X]^{\leqslant 2}$, $F(P)(\infty)$ denotes the degree-2 coefficient vector. For some vector polynomial $P_x \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^n$ such that $P_x(\infty) = x$, we thus obtain $F(P)(\infty) = F(x)$, which is the all-0 vector if and only if x is the MQ solution associated to F.

PIOP: simple version. We start with a simplified version of the PIOP underlying MQOM, which is depicted in Figure 1. The prover first picks random vector polynomials $P_x \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^n$ and $P_u \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^m$ such that $P_x(\infty) = x$ (while P_u is fully random). Then they send an oracle to these polynomials to the verifier. The prover further compute $P_\alpha = P_u + F(P_x)$ and sends it in clear (i.e. not as an oracle) to the verifier. The verifier samples a random point r in the evaluation domain Ω and queries the oracle to obtain the evaluations $P_x(r), P_u(r)$. They finally check that $P_\alpha(r) = P_u(r) + F(P_x(r))$.

The vector polynomial $P_z := F(P_x)$ is of degree 1 if and only if $P_z(\infty) = F(P_x(\infty)) = (0, \ldots, 0) \in \mathbb{K}^m$, meaning that $x = P_x(\infty)$ is a valid solution to the MQ instance defined by F. The soundness of this protocol follows from the Schwartz–Zippel lemma. Assume that $P_\alpha(r) = P_u(r) + F(P_x(r))$ holds for 3 different points of Ω, then because P_x, P_u, P_α are guaranteed to be of degree 1, we must have $P_\alpha = P_u + F(P_x)$ and hence $F(x) = 0$ (i.e. the prover indeed knows a solution x). Conversely, in the presence of a malicious prover (who does not know a right solution x), we can only have $P_\alpha(r) = P_u(r) + F(P_x(r))$ for at most two values r of Ω. The soundness error of the PIOP is hence of $2/|\Omega|$.

The zero-knowledge property of this protocol holds for two reasons. First, thanks to the addition of the random vector polynomial P_u, the vector polynomial P_α is further uniformly random. Then, any evaluations $P_x(r), P_u(r)$ are independent of x thanks to the randomness involved in these polynomials.

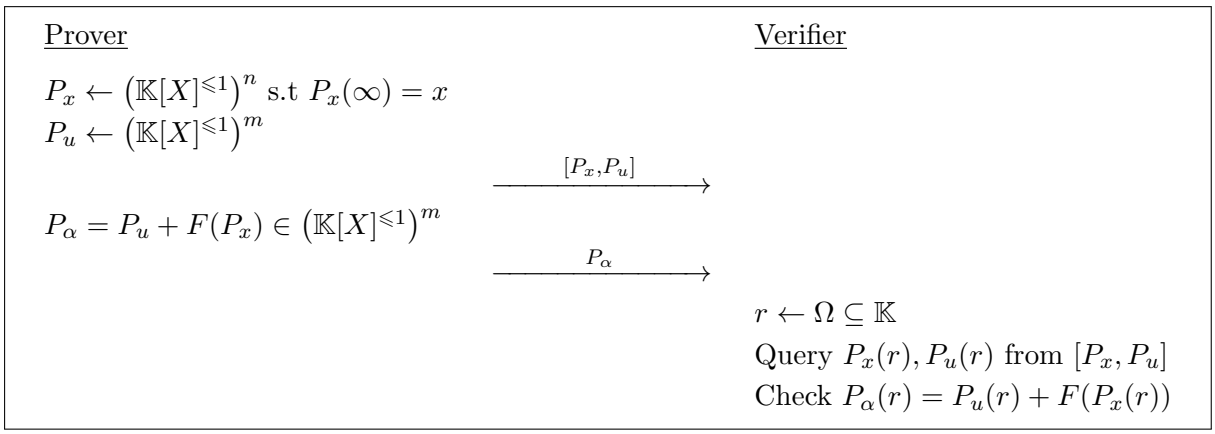


Figure 1: Polynomial IOP underlying MQOM (simple version).

Remark 1. *The original TCitH-Π_{PC} protocol does not encode the secret x as the leading coefficient of P_x but as its constant term (as for the original Shamir's secret sharing). While this choice has not impact on the soundness nor on the communication, choosing the leading term has some advantages in terms of computation (see Section 5 for further discussion).*

Besides the polynomial oracle, the communication cost of the above PIOP is the size of P_α which is made of $2m$ elements of the extension field $\mathbb{K}$. We now described two ways to batch the coordinates of P_α enabling to substantially lower this communication cost.

Batching with field embedding. The size of the vector polynomial P_α can be reduced by a factor μ, the extension degree of $\mathbb{K}/\mathbb{F}$. Let $\beta_1, \ldots, \beta_\mu$ an $\mathbb{F}$-basis of $\mathbb{K}$ and let ϕ the $\mathbb{F}$-linear field-embedding isomorphism:

$$\phi : (e_1, \ldots, e_\mu) \in \mathbb{F}^\mu \mapsto \sum_{i=1}^{\mu} e_i \cdot \beta_i \in \mathbb{K} \; . \tag{2}$$

In what follows, m is assumed to be a multiple of μ, which is always the case with our considered parameters. We further denote:

$$\Phi : (e_1, \ldots, e_m) \in \mathbb{F}^m \mapsto (\phi(e_1, \ldots, e_\mu), \phi(e_{\mu+1}, \ldots, e_{2\mu}), \ldots, \phi(e_{m-\mu}, \ldots, e_m)) \in \mathbb{K}^{\frac{m}{\mu}} \; . \tag{3}$$

Recall that the protocol aims to prove $P_z(\infty) = (0, \ldots, 0)$, where $P_z := F(P_x)$ and where $P_z(\infty)$ denotes the degree-2 coefficient vector of P_z. Although the polynomial P_x belongs to $\left(\mathbb{K}[X]^{\leqslant 1}\right)^n$, its leading term is defined as $P_x(\infty) = x \in \mathbb{F}^n$. Assume that $P_x(\infty)$ is ensured to belong to $\mathbb{F}^n$ by the polynomial oracle $[P_x, P_u]$ (this property is indeed ensured by the polynomial commitment described hereafter in Section 2.1.3). By definition of F, we then have $P_z(\infty) \in \mathbb{F}^m$ and hence:

$$P_z(\infty) = (0, \ldots, 0) \in \mathbb{F}^m \quad \iff \quad \Phi(P_z)(\infty) = (0, \ldots, 0) \in \mathbb{K}^{\frac{m}{\mu}} \; .$$

This means that P_α can be defined as

$$P_\alpha = P_u + \Phi(F(P_x)) \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^{\frac{m}{\mu}}$$

with $P_u \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^{\frac{m}{\mu}}$. Introducing this tweak in the above protocol does not change its soundness, thanks to the following equivalence:

$$\begin{aligned} P_\alpha \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^{\frac{m}{\mu}} &\iff \Phi(F(P_x)) \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^{\frac{m}{\mu}} \\ &\iff \Phi(F(P_x))(\infty) = (0,\ldots,0) \in \mathbb{K}^{\frac{m}{\mu}} \\ &\iff F(P_x)(\infty) = F(x) = (0,\ldots,0) \in \mathbb{F}^m \end{aligned}$$

Batching with random combinations. To further reduce the size of P_α, we can use the standard approach to batch the verification of several relations using random linear combinations. In our context, this means batching the m/μ coordinates of $\Phi(F(P_x))$ into η random linear combinations for some parameters $\eta \in \mathbb{N}$. Let $\Gamma \in \mathbb{K}^{\eta \times \frac{m}{\mu}}$, a matrix randomly sampled by the verifier. The prover now define P_α as:

$$P_\alpha = P_u + \Gamma \cdot \Phi(P_z) \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^\eta$$

with $P_u \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^\eta$ and $P_z = F(P_x) \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^m$. We then have:

$$\Pr\left[\Gamma \cdot \Phi(P_z)(\infty) = (0,\ldots,0) \mid \Phi(P_z)(\infty) \neq (0,\ldots,0)\right] \;\leq\; \frac{1}{|\mathbb{K}|^\eta}\,.$$

For a target λ-bit security, selecting $\eta := \lceil \lambda / \log_2 |\mathbb{K}| \rceil$ makes the above soundness error $\leq 2^{-\lambda}$.

PIOP: full version. Figure 2 provides the description of the PIOP integrating the two subsequent batching strategies. While MQOM always relies on the field-embedding batching, the random-combination batching is optional. When disabled, the PIOP skips the step of sampling Γ by the verifier and sending it to the prover (represented between dashed lines on Figure 2). Γ is then simply defined as the identity matrix $\Gamma = I_\eta \subseteq \mathbb{K}^{\eta \times \eta}$ with $\eta = m/\mu$.

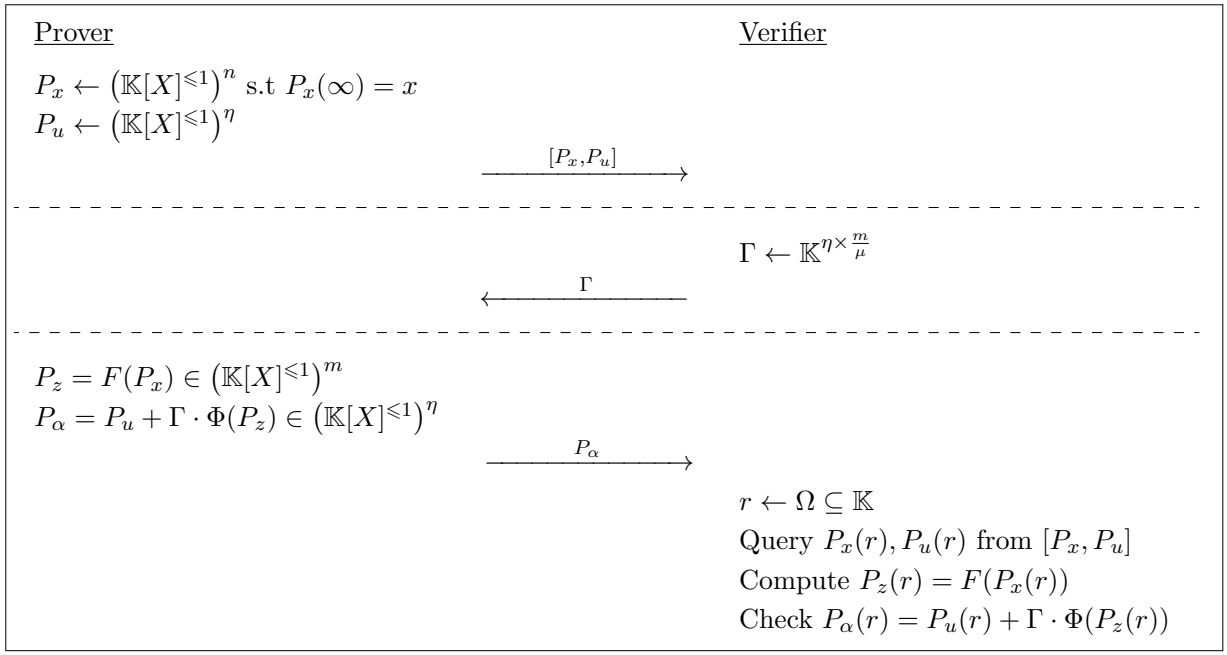


Figure 2: Polynomial IOP underlying MQOM (full version).

Disabling random-combination batching makes the MQOM zero-knowledge proof-of-knowledge a simple sigma protocol. This comes at a moderate communication cost (depending of the parameters) thanks to the field-embedding batching which significantly reduces the communication overhead in the context. On the other hand, when random-combination batching is enabled, the main purpose of the field-embedding batching is to reduce the computation overhead. More precisely, we could skip the application of Φ and sample a larger matrix $\Gamma \in \mathbb{K}^{\eta \times m}$. While this would not change the communication or the soundness of the PIOP, this would make sampling Γ as well as computing the product $\Gamma \cdot P_z$ heavier.

2.1.3 Line commitment scheme

To compile the above polynomial IOP into a zero-knowledge proof of knowledge (ZK-PoK), the polynomial oracle is replaced by a polynomial commitment scheme. This means that the prover sends a commitment $\mathsf{Com}(P_x, P_u)$ to the polynomials P_x, P_u in place of the oracle. Later on, the query from the verifier to the oracle is replaced by an evaluation opening protocol:

1. the verifier sends the evaluation point r to the prover,
2. the prover replies with evaluations $v_x = P_x(r), v_u = P_u(r)$ along with an opening proof π,
3. the verifier checks π and, in case of success, accepts v_x, v_u as valid evaluations.

The commitment scheme should be:

- *binding*: the commitment $\mathsf{Com}(P_x, P_u)$ defines unique polynomials P_x, P_u and it should be hard for a mallicious prover to later come up with evaluations v_x, v_u and an opening proof π passing the verification while $v_x \neq P_x(r)$ or $v_u \neq P_u(r)$;
- *hiding/zero-knowledge*: the verifier does not learn anything more than v_x, v_u about P_x, P_u.

In the context of this specification, the polynomials P_x and P_u are limited to be of degree 1. In consequence, we shall use the terminology of *line commitment scheme*. We explain the line commitment scheme used in MQOM hereafter. This construction relies on a GGM seed tree.

GGM seed tree. A GGM seed tree consists in pseudorandomly expanding a root seed `rseed` into N leaf seeds `lseed[0]`, ..., `lseed[`$N-1$`]` using a binary tree. The process is summarized by

$$\begin{cases} \texttt{node[1]} \leftarrow \texttt{rseed} \\ (\texttt{node[}2i\texttt{]}, \texttt{node[}2i+1\texttt{]}) \leftarrow \mathsf{SeedDerive}(\texttt{node[}i\texttt{]}) \quad \text{for } 1 \leq i \leq N-1 \end{cases} \tag{4}$$

where SeedDerive is a seed derivation function. The leaf seeds are then defined as the last N nodes, namely `lseed[`i`]` := `node[`$N+i$`]` for all $i \in [0, N-1]$. The structure of a GGM seed tree and underlying node numbering are illustrated on Figure 3. In the scope of the present specification, the number of leaves N is always a power of two.

The GGM tree structure enables to reveal all-but-one leaf seeds from $\log_2(N)$ nodes of the tree. Let $i^* \in [0, N-1]$ be the index of the leaf seed that should remain hidden. Any node on the path from the hidden leaf `lseed[`i^*`]` = `node[`$N+i^*$`]` to the root of the tree should remain hidden (since `lseed[`i^*`]` can be derived from any of those nodes). The indexes of the node on this hidden path belong to the following set:

$$\mathcal{H} = \{\lfloor (N+i^*)/2^j \rfloor \mid 0 \leq j \leq \log_2(N) - 1\} \ .$$

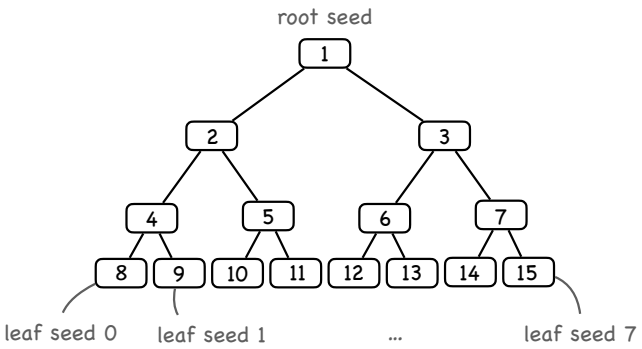


Figure 3: GGM tree structure and node numbering.

On the other hand, the *sibling path*, which is made of the siblings of the hidden nodes, can be revealed without giving any information $\texttt{lseed}[i^*]$. According to the binary tree structure, the sibling of $\texttt{node}[i]$ is $\texttt{node}[i \oplus 1]$. The sibling path of $\texttt{lseed}[i^*]$ is hence defined as:

$$\texttt{path} = \{\texttt{node}[i \oplus 1] \mid i \in \mathcal{H}\} \ .$$

By running the tree derivation of Equation 4 from all the seeds in the sibling path, one reconstructs a partial tree composed of all the nodes but the hidden path. In particular, this partial tree includes all the leaves but the hidden leaf $\texttt{lseed}[i^*]$.

GGM seed trees have been introduced in the context of MPC-in-the-Head to commit additive secret sharings with efficient opening of all-but-one shares [KKW18]. From a random root seed `rseed`, one expands a GGM seed tree to obtain the leaf seeds $\texttt{lseed}[0], \ldots, \texttt{lseed}[N-1]$. Then, each leaf seed is expanded into a share:

$$\bar{x}_i \in \mathbb{F}^n \leftarrow \mathsf{PRG}(\texttt{lseed}[i]) \ ,$$

and a correction value Δ_x is defined as:

$$\Delta_x = x - \sum_{i=0}^{N-1} \bar{x}_i \ . \tag{5}$$

By definition, $(\bar{x}_0 + \Delta_x)$, $\bar{x}_1$, $\ldots$, $\bar{x}_{N-1}$ forms an additive secret sharing of x. This additive sharing is committed by deriving a commitment for each seed:

$$\texttt{ls\_com}[i] \leftarrow \mathsf{SeedCommit}(\texttt{lseed}[i])$$

and sending a global commitment:

$$\texttt{com} \leftarrow \mathsf{Commit}(\texttt{ls\_com}[0], \ldots, \texttt{ls\_com}[N-1], \Delta_x)$$

to the verifier. Later on, the verifier can challenge the prover to open all the additive shares of x but one, say the share of index i^*. The prover then reveals the sibling path of $\texttt{lseed}[i^*]$, the correction value Δ_x and the commitment $\texttt{ls\_com}[i^*]$. From all the leaf seeds (but $\texttt{lseed}[i^*]$), the verifier can expand all the shares $\bar{x}_i$ (but $\bar{x}_{i^*}$) and correct $\bar{x}_0$ with Δ_x. The verifier can also recompute the commitments $\texttt{ls\_com}[i]$, for all $i \neq i^*$, and together with $\texttt{lseed}[i^*]$ and Δ_x, recompute and check the global commitment.

Line commitment from GGM seed tree. As described in the TCitH framework [FR23b], one can use a sharing conversion technique from [CDI05] to turn an all-but-one additive secret sharing (a.k.a. *replicated secret sharing*) into a Shamir's secret sharing, with underlying polynomial P_x encoding x. Then, revealing all-but-one additive shares of x amounts to revealing one Shamir's share of x, i.e., one evaluation of the polynomial P_x. A similar technique was also previously described in the VOLE-in-the-Head framework [BBD+23] to commit small VOLE correlations based on the small-field VOLE construction from [Roy22].

The committed degree-1 polynomial (or line) is defined as:

$$P_x = \Delta_x P_0 + \sum_{i=0}^{N-1} \bar{x}_i P_i \ \in \mathbb{K}[X]^{\leqslant 1} \tag{6}$$

where $P_0, \ldots, P_{N-1} \in \mathbb{K}[X]^{\leqslant 1}$ are fixed degree-1 polynomials defined as:

$$\begin{cases} P_i(\omega_i) = 0 \\ P_i(\infty) = 1 \end{cases} \tag{7}$$

for some predefined evaluation points $\Omega = \{\omega_0, \ldots, \omega_{N-1}\} \subseteq \mathbb{K}$. From this definition, we have that P_x encodes the secret x by:

$$P_x(\infty) = \Delta_x + \sum_{i=0}^{N-1} \bar{x}_i = x \ .$$

Moreover, the evaluation of P_x in a point $\omega_{i^*} \in \Omega$ can be derived from the all-but-one additive sharing of index i^*. Namely, from all the $\bar{x}_i$ but $\bar{x}_{i^*}$, the verifier can recompute:

$$P_x(\omega_{i^*}) = \Delta_x P_0(\omega_{i^*}) + \sum_{i=0}^{N-1} \bar{x}_i P_i(\omega_{i^*}) = \Delta_x P_0(\omega_{i^*}) + \sum_{i \neq i^*} \bar{x}_i P_i(\omega_{i^*})$$

where the above equality holds because $P_{i^*}(\omega_{i^*}) = 0$. Since any other evaluation of P_x would require the knowledge of $\bar{x}_{i^*}$, the verifier only learns $P_x(\omega_{i^*})$ from the all-but-one opening.

Following Equation 6 and Equation 7, and assuming $\omega_0 = 0$, we have $P_i(X) = X - \omega_i$ for all $i \in [0, N-1]$, and:

$$P_x(X) = x \cdot X - \sum_{i=0}^{N-1} \omega_i \cdot \bar{x}_i \ .$$

Remark 2. *As mentionned in Remark 1, the original TCitH scheme [FR23b] does not encode the secret x as $P_x(\infty)$ but as $P_x(0)$ (as in the original Shamir's secret sharing). For this reason, the P_i polynomials are defined such that $P_i(\omega_i) = 0$ and $P_i(0) = 1$ in [FR23b]. In the present specification, we choose to encode x in $P_x(\infty)$ which offers some advantages, as discussed in Section 5.*

Committing the random polynomial P_u works the same way but without correction value. Namely, the additive shares $\bar{u}_i \in \mathbb{K}^\eta$ are also pseudorandomly sampled from the leaf seeds `lseed[`i`]` for all $i \in [0, N-1]$ and P_u is defined as:

$$P_u(X) = \sum_{i=0}^{N-1} \bar{u}_i P_i = \Big(\sum_{i=0}^{N-1} \bar{u}_i \Big) \cdot X - \sum_{i=0}^{N-1} \omega_i \cdot \bar{u}_i \ \in \big(\mathbb{K}[X]^{\leqslant 1}\big)^\eta \ .$$

Correlated tree optimization. The correlated half-tree technique introduced in [GYW+23] enables to slightly reduce the communication cost of a GGM all-but-one sharing commitment. For a fixed $\delta \in \{0,1\}^\lambda$ (where $\{0,1\}^\lambda$ is the definition space of the seeds), the correlated tree technique maintains the following invariant. At any given level in the tree, the XOR of all the seeds equal δ. To enforce this property, the definition of the tree is modified as follows:

$$\begin{cases} \texttt{node}[2] \leftarrow \texttt{rseed} \\ \texttt{node}[3] \leftarrow \texttt{rseed} \oplus \delta \end{cases} \quad \text{and} \quad \begin{cases} \texttt{node}[2i] \leftarrow \mathsf{SeedDerive}(\texttt{node}[i]) \\ \texttt{node}[2i+1] \leftarrow \texttt{node}[2i] \oplus \texttt{node}[i] \end{cases}$$

for $1 \leq i \leq N-1$. One can indeed check that, for any $j \geq 1$, we thus get:

$$\begin{aligned} \delta &= \texttt{node}[2] \oplus \texttt{node}[3] \\ &= \texttt{node}[4] \oplus \texttt{node}[5] \oplus \texttt{node}[6] \oplus \texttt{node}[7] \\ &\vdots \\ &= \bigoplus_{i=2^j}^{2^{j+1}-1} \texttt{node}[i] \ . \end{aligned}$$

Then redefining:

$$\bar{x}_i \in \mathbb{F}^n \leftarrow \texttt{lseed}[i] \parallel \mathsf{PRG}(\texttt{lseed}[i]) \ ,$$

we get that the λ first bits of $\sum_{i=0}^{N-1} \bar{x}_i$ equal δ (assuming that $\mathbb{F}$ is a binary field so that the field addition matches the XOR). By defining δ as the λ first bit of x, Equation 5 then implies that the λ first bits of Δ_x equal 0^λ (the all-0 λ-bit string). We can thus save λ bits in the communication of Δ_x.

Wrapping up. When plugging the above line commitment scheme into the PIOP of Figure 2, we obtain the MQOM ZK PoK depicted in Figure 4. We note that this PoK is extractable knowledge sound under an idealized assumption on the SeedCommit primitive (e.g. in the random oracle or ideal cipher model). This leads to a tight EUF-CMA security since the reduction can extract the secret from any valid commitment.

2.1.4 Compilation to signature scheme

The MQOM signature scheme is constructed in two steps: first, we amplify the soundness of the MQOM ZK PoK (Figure 4) through parallel repetition; then, we apply the Fiat-Shamir transform to render it non-interactive and message-bound. In addition, we introduce some tweaks to lower the signature size and improve security and performances.

Parallel repetitions. The MQOM ZK PoK (Figure 4) has round-by-round soundness with soundness error $\epsilon_1 = 1/|\mathbb{K}|^\eta$ for the (optional) batching round and $\epsilon_2 = 2/|\Omega| = 2/N$ for the next round (see Section 2.1.2). To achieve a soundness error below $2^{-\lambda}$ for a target security of λ bits, η is fixed to $\eta = \lambda / \log_2 |\mathbb{K}|$ (the result of this division is always an integer for our considered parameters). For the next round, we rely on parallel repetition. Namely, the protocol is repeated τ times to make $(2/N)^\tau$ sufficiently small.

Specifically, the considered line commitment is turned into a *batch line commitment* (BCL), which commits τ pairs of polynomials $(P_x^{(0)}, P_u^{(0)}), \ldots, (P_x^{(\tau-1)}, P_u^{(\tau-1)})$. For each of them, a polynomial $P_\alpha^{(e)} = P_u^{(e)} + \Gamma \cdot \Phi(F(P_x^{(e)}))$ is computed and sent to the verifier, where (in case

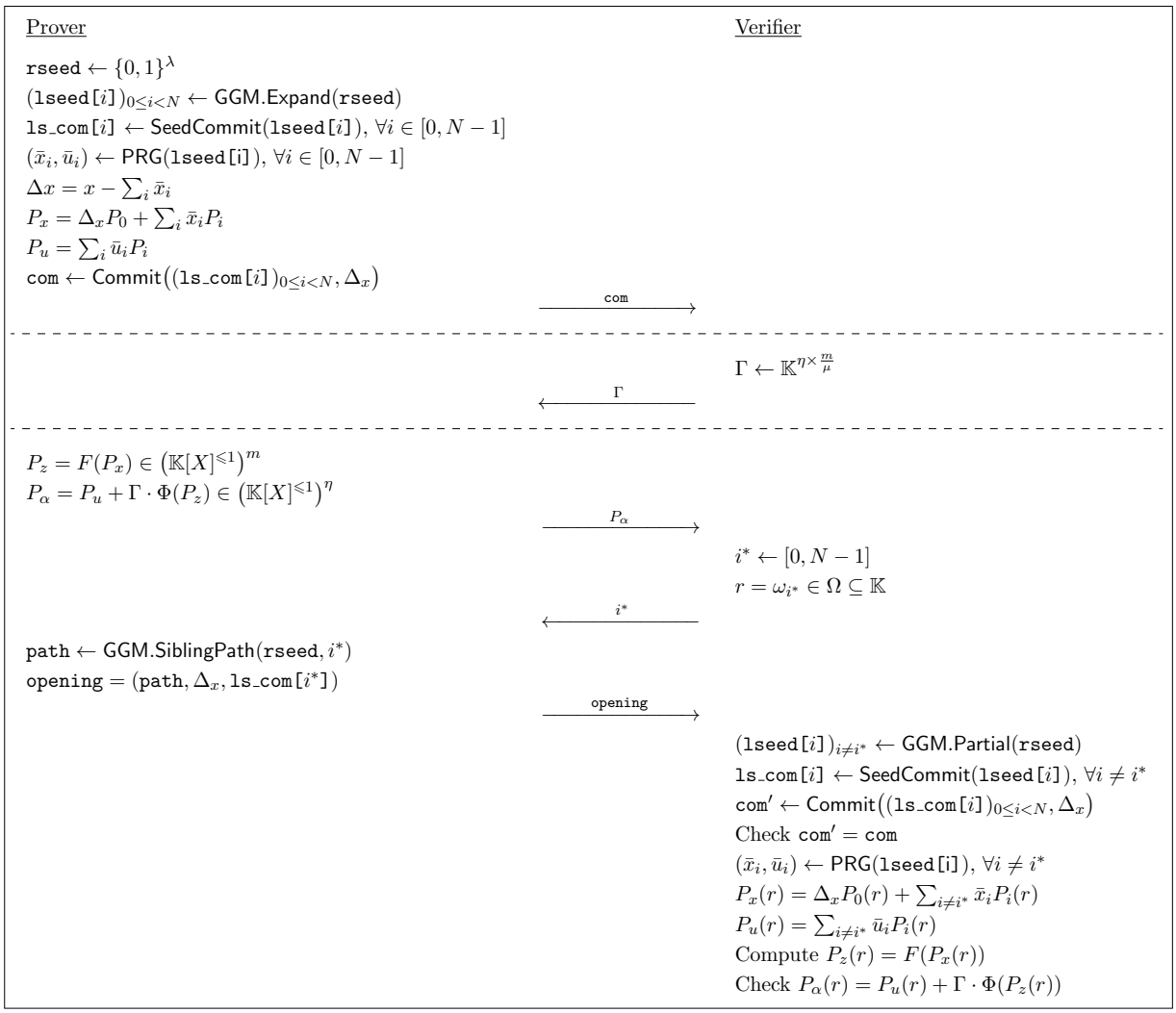


Figure 4: MQOM zero-knowledge proof of knowledge.

of batching) the matrix Γ is the same for each repetition. The verifier then pick τ indexes $i^*[0], \ldots, i^*[\tau-1]$, each leading to an evaluation point $r^{(e)} \in \Omega$. The prover finally reveal the opening data for the verifier to check and compute the evaluations $P_x^{(e)}(r^{(e)}), P_u^{(e)}(r^{(e)})$.

Concretely, the BCL scheme relies on τ parallel GGM trees, each giving rise to its own set of leaf seeds $(\texttt{lseed}[e][i])_{0\leq i<N}$, corresponding commitments, $(\texttt{ls\_com}[e][i])_{0\leq i<N}$, and correction value $\Delta_x[e]$, for $e \in [0, \tau-1]$. The global commitment is then defined as:

$$\texttt{com} \leftarrow \mathsf{Commit}(c^{(0)}, \ldots, c^{(\tau-1)}, \Delta_x[0], \ldots, \Delta_x[\tau-1])$$

where $c^{(e)} \leftarrow \mathsf{Commit}(\texttt{ls\_com}[e][0], \ldots, \texttt{ls\_com}[e][\tau-1])$. Finally, the global opening consists of the opening tuple $(\texttt{path}[e], \Delta_x[e], \texttt{ls\_com}[e][i^*[e]])$ for each execution $e \in [0, \tau-1]$.

Hash commitment of P_α. To reduce the communication of the ZK PoK protocol, a standard optimization is to send a hash commitment of the polynomials $P_\alpha^{(0)}, \ldots, P_\alpha^{(\tau-1)}$ instead of transmitting them in full. Doing so, the prover only needs to send the leading coefficient $\alpha_1^{(e)} \in \mathbb{K}^\eta$ for each of these polynomial rather than the full pair of coefficients $(\alpha_0^{(e)}, \alpha_1^{(e)}) \in (\mathbb{K}^\eta)^2$. From the

open evaluations $P_x^{(e)}(r^{(e)})$, $P_u^{(e)}(r^{(e)})$, the verifier deduces $P_\alpha^{(e)}(r^{(e)})$ an compute a candidate value for the constant term:

$$\alpha_0^{(e)} = P_\alpha^{(e)}(r^{(e)}) - \alpha_0^{(e)} \cdot r^{(e)} \ .$$

Finally, the verifier checks this against the hash commitment. This technique effectively halves the communication cost (or signature footprint) associated with the polynomials $P_\alpha^{(0)}, \ldots, P_\alpha^{(\tau-1)}$.

Fiat-Shamir transform. In the following, we shall denote by $\mathtt{com}_1$ the BLC commitment and $\mathtt{com}_2$ the hash commitment of the polynomials $P_\alpha^{(0)}, \ldots, P_\alpha^{(\tau-1)}$ following the above optimization.

In the first (optional) round, the application of Fiat-Shamir consists in deriving the matrix $\Gamma \in \mathbb{K}^{\eta \times \frac{m}{\mu}}$ from the commitment by a call to an extendable output hash function:

$$\Gamma \leftarrow \mathsf{XOF}(\mathtt{com}_1) \ .$$

In the second round, the application of Fiat-Shamir consists in deriving the challenge indexes $i^*[0], \ldots, i^*[\tau-1]$ from the commitments $\mathtt{com}_1$ and $\mathtt{com}_2$. We further input the message as well as the public key is this hash computation, which gives:

$$(i^*[0], \ldots, i^*[\tau-1]) \leftarrow \mathsf{XOF}(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2, \mathtt{msg}) \ .$$

Grinding. To reduce the number of parallel repetitions, we employ *grinding*. Namely, we include a w-bit proof-of-work increasing the number of iterations of the second hash by a factor of 2^w on average for a grinding parameter w. The number of repetitions is then relaxed to satisfy

$$\left(\frac{2}{N}\right)^\tau \leq \frac{1}{2^{\lambda-w}} \ , \tag{8}$$

namely, we fix $\tau := \lceil (\lambda - w)/(\log_2(N) - 1) \rceil$.

The second Fiat-Shamir hash is then performed as follows. One first compute a hash value

$$\mathtt{hash} \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2, \mathtt{msg})$$

and then iterates

$$(i^*, \mathtt{val}) \leftarrow \mathsf{XOF}(\mathtt{hash}, \mathtt{nonce})$$

where $i^* = (i^*[0], \ldots, i^*[\tau-1]) \in [0, N-1]^\tau$, $\mathtt{val} \in [0, 2^w - 1]$, and $\mathtt{nonce} \in [0, 2^{32} - 1]$ a 32-bit counter. The above computation is repeated by increasing $\mathtt{nonce}$ until obtaining $\mathtt{val} = 0$. The succeeding $\mathtt{nonce}$ value is included into the signature so that the verifier only needs to run the right XOF computation once. The verifier further checks that the XOF output satisfies $\mathtt{val} = 0$ to accept the signature.

In the random oracle model (ROM), any attempt to forge a signature requires the adversary to make a random oracle query to get the associated challenge i^*. Using grinding, each such random oracle query has only a probability 2^{-w} to yield a valid grinding value $\mathtt{val} = 0$. As a result, the (amplified) second-round soundness error scales from $\epsilon_1 = (2/N)^\tau$ down to $\epsilon_1 \cdot 2^{-w}$ from which we get the relaxation of Equation 8. The reader is referred to [Sta21] for a formal argument.

Seed derivation and commitment in GGM trees. The functions SeedDerive, SeedCommit, and PRG are defined based on a block cipher:

$$\mathsf{Enc} : (\mathtt{key}, \mathtt{ptx}) \in \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \mathtt{ctx} \in \{0,1\}^\lambda \ , \tag{9}$$

which is instantiated as AES-128 (for $\lambda = 128$) or Rijndael-256-256 (for $\lambda = 192$ and $\lambda = 256$). This choice ensures efficiency by leveraging the AES hardware instructions available on modern CPUs.

For security, the derivation, commitment and expansion of seeds must incorporate a random salt, which is sampled at the beginning of the signing process and included as part of the signature. Since seeds are only λ bits long, using this salt mitigates the risk of collisions in tree derivation and prevents accelerated exhaustive searches for hidden seeds. Additionally, this salt is modified to enforce domain separation between different calls to SeedDerive and SeedCommit within a single signature computation.

We employ a Davies-Meyer construction to transform the block cipher Enc into the seed derivation and commitment primitives. Here, the salt acts as the key, while the seed is used as the plaintext and also introduced in the output through a feed-forward mechanism. Given the XOR-invariant properties of correlated GGM tree optimizations, a simple XOR-based feed-forward would make the seed derivation process invertible. Instead, as suggested in [GKW+20], we use a $\mathbb{F}_2$-linear orthomorphism ψ, defined as:

$$\psi : (x_l \parallel x_r) \in \{0,1\}^\lambda \mapsto (x_l \oplus x_r \parallel x_l) \ . \tag{10}$$

Concretely, we define encryption with feed-forward (EncFF) as follows:

$$\mathsf{EncFF} : (\mathsf{seed}, \mathtt{tweak}) \mapsto \mathsf{Enc}(\mathtt{salt} \oplus \mathtt{tweak}, \mathsf{seed}) \oplus \psi(\mathsf{seed}) \ . \tag{11}$$

This allows us to define the seed derivation function as:

$$\mathsf{SeedDerive}(\mathsf{seed}, \mathtt{tweak}) = \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak}) \ , \tag{12}$$

and the seed commitment function as:

$$\mathsf{SeedCommit}(\mathsf{seed}, \mathtt{tweak}) = \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak}) \parallel \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak} \oplus 1) \ . \tag{13}$$

The domain-separator tweaks are defined to ensure security. For seed derivation, we assign a unique $\mathtt{tweak}$ for each pair (e, j), where $e \in [0, \tau - 1]$ denotes the execution index, and $j \in [0, \log_2(N) - 1]$ represents the height of the current node in the tree. This guarantees that any two seeds in the revealed sibling paths result from distinct derivation functions, preventing accelerated preimage attacks. For commitment, we use a distinct pair $(\mathtt{tweak}, \mathtt{tweak} \oplus 1)$ per execution index $e \in [0, \tau - 1]$ (and which are also disjoint from the seed derivation tweaks), ensuring that hidden seed commitments are derived from separate commitment functions, thereby thwarting accelerated preimage attacks.

2.2 Notations

Mathematical notations. We summarize the mathematical notations used in the algorithmic description of MQOM in Table 1. The concrete instantiations of $\mathbb{F}$ and $\mathbb{K}$, with underlying irreducible polynomials, the evaluation points $\omega_0, \ldots, \omega_{N-1}$, and the $\mathbb{F}$-basis of $\mathbb{K}$, $\beta_1, \ldots, \beta_\mu$, are defined in Section 3.

Table 1: Mathematical notations.

$\mathbb{F}$	The base field, a finite field of characteristic 2
$\mathbb{K}$	The extension field, a finite extension of $\mathbb{F}$
$\Omega = \{\omega_0, \ldots, \omega_{N-1}\}$	The evaluation domain, a subset of $\mathbb{K}$
$\{\beta_1, \ldots, \beta_\mu\}$	An $\mathbb{F}$-basis of $\mathbb{K}$
$\mathbb{F}[X]^{\leqslant 1}$	Set of polynomials of degree ≤ 1 with coefficients from $\mathbb{F}$
$\mathbb{K}[X]^{\leqslant 1}$	Set of polynomials of degree ≤ 1 with coefficients from $\mathbb{K}$
ϕ	Field-embedding isomorphism $\phi : \mathbb{F}^\mu \to \mathbb{K}$
Φ	Block-wise field-embedding isomorphism $\phi : \mathbb{F}^m \to \mathbb{K}^{\frac{m}{\mu}}$
ψ	Linear orthomorphism $\psi : \{0,1\}^\lambda \to \{0,1\}^\lambda$
$\lvert\cdot\rvert_2$	Bit-size of a field-element vector
$\lvert\cdot\rvert_8$	Byte-size of a field-element vector
I_η	Identity matrix on $\mathbb{K}^{\eta\times\eta}$
0^ℓ	All-0 ℓ-bit string

We recall the definition of the three morphisms ϕ, Φ and ψ. For $\beta_1, \ldots, \beta_\mu$ an $\mathbb{F}$-basis of $\mathbb{K}$, we let ϕ be the $\mathbb{F}$-linear field-embedding isomorphism:

$$\phi : (e_1, \ldots, e_\mu) \in \mathbb{F}^\mu \mapsto \sum_{i=1}^{\mu} e_i \cdot \beta_i \in \mathbb{K} \ .$$

The block-wise field-embedding isomorphism Φ is then defined as:

$$\Phi : (e_1, \ldots, e_m) \in \mathbb{F}^m \mapsto (\phi(e_1, \ldots, e_\mu), \phi(e_{\mu+1}, \ldots, e_{2\mu}), \ldots, \phi(e_{m-\mu}, \ldots, e_m)) \in \mathbb{K}^{\frac{m}{\mu}} \ . \quad (14)$$

The $\mathbb{F}_2$-linear orthomorphism ψ is defined as:

$$\psi : (x_l \parallel x_r) \in \{0,1\}^\lambda \mapsto (x_l \oplus x_r \parallel x_l) \ . \quad (15)$$

Notations for MQOM parameters. The notations for the various parameters of MQOM are summarized in Table 2. The specific instantiations of these parameters, which define the different instances of MQOM, are provided in Section 3.

Notations for algorithmic description. The variables used in the algorithmic description of MQOM, along with their respective definition domains, are summarized in Table 3.

Table 2: Parameters of the MQOM signature scheme.

MQ parameters:	
$\mathbb{F}$	Base field
n	Number of unknowns
m	Number of equations
Proof system parameters:	
λ	Security parameter
μ	Extension degree $\mu = [\mathbb{K} : \mathbb{F}]$
N	Size of the evaluation domain $N = \lvert\Omega\rvert$
η	Number of internal repetitions of the proof system
τ	Number of external repetitions of the proof system
w	Grinding proof-of-work parameter

Table 3: Notations of the MQOM signature scheme.

x	Secret MQ solution	$\mathbb{F}^n$
A_i	Quadratic-part matrix of the i-th MQ equation	$\mathbb{F}^{n\times n}$
b_i	Linear-part vector of the i-th MQ equation	$\mathbb{F}^n$
y_i	Constant part of the i-th MQ equation	$\mathbb{F}$
`seed_key`	Master seed for key generation	$\{0,1\}^{2\lambda}$
`seed_eq`$[i-1]$	Seed for MQ equations $\{A_i\}$, $\{b_i\}$	$\{0,1\}^{2\lambda}$
$\texttt{com}_1$	BLC commitment	$\{0,1\}^{2\lambda}$
$\texttt{com}_2$	Hash commitment $\texttt{com}_2 = \mathsf{Hash}(\alpha_0, \alpha_1)$	$\{0,1\}^{2\lambda}$
`key`	BLC opening key $\texttt{key} = (\texttt{node}, \Delta_{x'})$	–
`opening`	BLC opening $\texttt{opening} = (\texttt{path}, \texttt{out\_ls\_com}, \Delta_{x'})$	–
i^*	Hidden-leaf indexes $i^* = (i^*[0], \ldots, i^*[\tau-1])$	$[0, N-1]^\tau$
`nonce`	Grinding nonce	$[0, 2^{32}-1]$
`val`	Grinding test value	$[0, 2^w-1]$
`batching`	Boolean for enabling / disabling the batching variant	$\{\mathsf{True}, \mathsf{False}\}$
x_0	Coefficient array $x_0 = (x_0[0], \ldots, x_0[\tau-1])$	$\left(\mathbb{K}^n\right)^\tau$
u_0	Coefficient array $u_0 = (u_0[0], \ldots, u_0[\tau-1])$	$\left(\mathbb{K}^\eta\right)^\tau$
u_1	Coefficient array $u_1 = (u_1[0], \ldots, u_1[\tau-1])$	$\left(\mathbb{K}^\eta\right)^\tau$
α_0	Coefficient array $\alpha_0 = (\alpha_0[0], \ldots, \alpha_0[\tau-1])$	$\left(\mathbb{K}^\eta\right)^\tau$
α_1	Coefficient array $\alpha_1 = (\alpha_1[0], \ldots, \alpha_1[\tau-1])$	$\left(\mathbb{K}^\eta\right)^\tau$
`x_eval`	Evaluation array $\texttt{x\_eval} = (\texttt{x\_eval}[0], \ldots, \texttt{x\_eval}[\tau-1])$	$(\mathbb{K}^n)^\tau$
`u_eval`	Evaluation array $\texttt{u\_eval} = (\texttt{u\_eval}[0], \ldots, \texttt{u\_eval}[\tau-1])$	$(\mathbb{K}^\eta)^\tau$
`mseed`	Signature master seed	$\{0,1\}^\lambda$
`salt`	Signature salt	$\{0,1\}^\lambda$
`rseed`	Array $\texttt{rseed} = (\texttt{rseed}[0], \ldots, \texttt{rseed}[\tau-1])$	–
`rseed`$[e]$	GGM root seed for execution e	$\{0,1\}^\lambda$
`lseed`	Array $\texttt{lseed} = (\texttt{lseed}[0], \ldots, \texttt{lseed}[\tau-1])$	–
`lseed`$[e]$	Array $\texttt{lseed}[e] = (\texttt{lseed}[e][0], \ldots, \texttt{lseed}[e][N-1])$	–
`lseed`$[e][i]$	Leaf seed of index i for execution e	$\{0,1\}^\lambda$
`ls_com`	Array $\texttt{ls\_com} = (\texttt{ls\_com}[0], \ldots, \texttt{ls\_com}[\tau-1])$	–
`ls_com`$[e]$	Array $\texttt{ls\_com}[e] = (\texttt{ls\_com}[e][0], \ldots, \texttt{ls\_com}[e][N-1])$	–
`ls_com`$[e][i]$	Leaf seed commitment of index i for execution e	$\{0,1\}^{2\lambda}$
`node`	Array $\texttt{node} = (\texttt{node}[0], \ldots, \texttt{node}[\tau-1])$	–
`node`$[e]$	Array $\texttt{node}[e] = (\texttt{node}[e][2], \ldots, \texttt{node}[e][2N+1])$	–
`node`$[e][j]$	Node seed of index j for execution e	$\{0,1\}^\lambda$
`path`	Array $\texttt{path} = (\texttt{path}[0], \ldots, \texttt{path}[\tau-1])$	–
`path`$[e]$	Array $\texttt{path}[e] = (\texttt{path}[e][0], \ldots, \texttt{path}[e][\log_2(N)-1])$	–
`path`$[e][j]$	Sibling-path seed of index j (from leaf to root) for execution e	$\{0,1\}^\lambda$
`exp`$[e][i]$	Expanded leaf seed	$\{0,1\}^{\lvert x\rvert_2+\lvert u\rvert_2-\lambda}$

2.3 Data representation

The elementary data type in MQOM is a byte string. Any other data types, such as bit-strings, vectors of field elements, tuples or arrays, are serialized and represented as byte strings in input and output of the MQOM algorithms. We detail hereafter how these data types are serialized into byte strings.

In the algorithms presented in the following sections, we use the Serialize routine to explicitly apply the serialization process depicted below. On the other hand, the Parse routine peforms the inverse operation with the output format implicit from the context.

Bit-string. A bit-string is an element from $\{0,1\}^\ell$ for some $\ell \in \mathbb{N}$ (most of the time $\ell = \lambda \in \{128, 192, 256\}$ or $\ell = 2\lambda \in \{256, 384, 512\}$). A bit-string is naturally represented as a byte-string of size $\lceil \ell/8 \rceil$. Whenever ℓ is not a multiple of 8, the last ($\ell \bmod 8$) bits of the bit-string are the least significant bits in the last byte.

The different variants of MQOM involve three different fields: $\mathbb{F}_2$, $\mathbb{F}_{256}$ and $\mathbb{F}_{2^{16}}$. Below, we describe the serialization process for vectors of field elements for each of these fields.

Vectors of $\mathbb{F}_2$-elements. An element of $\mathbb{F}_2$ is naturally represented on a single bit. Vectors from $\mathbb{F}_2^\ell$ are only serialized for ℓ a multiple of 8. A vector $v = (v_1, \ldots, v_\ell) \in \mathbb{F}_2^\ell$ is serialized as:

$$\mathsf{Serialize}(v) = \mathsf{B}(v_1, \ldots, v_8) \parallel \ldots \parallel \mathsf{B}(v_{\ell-7}, \ldots, v_\ell)$$

where B is the byte-grouping function defined as:

$$\mathsf{B}(b_0, \ldots, b_7) = \sum_{i=0}^{7} 2^i \cdot \mathsf{int}(b_i)$$

with int the natural mapping $\mathbb{F}_2 \to \{0,1\} \subseteq \mathbb{N}$.

Vectors of $\mathbb{F}_{256}$-elements. An element of $\mathbb{F}_{256}$ is naturally represented as a byte. Specifically, an element $e \in \mathbb{F}_{256}$ is represented as a tuple $(e_0, \ldots, e_7) \in \mathbb{F}_2^8$ such that $e = \sum_{i=0}^{7} e_i \cdot \xi^i$ for ξ a primitive element of $\mathbb{F}_{256}$ over $\mathbb{F}_2$ (i.e. $\mathbb{F}_{256} = \mathbb{F}_2[\xi]$). The byte representation of such an element is defined as:

$$\mathsf{byte}(e) = \mathsf{B}(e_0, \ldots, e_7) \quad \Leftrightarrow \quad e = \sum_{i=0}^{7} e_i \cdot \xi^i \ .$$

A vector $(v_1, \ldots, v_\ell) \in \mathbb{F}_{256}^\ell$ is naturally serialized as:

$$\mathsf{Serialize}(v) = \mathsf{byte}(v_1) \parallel \ldots \parallel \mathsf{byte}(v_\ell) \ .$$

Vectors of $\mathbb{F}_{2^{16}}$-elements. An element $e \in \mathbb{F}_{2^{16}}$ is represented as a pair $(e_0, e_1) \in \mathbb{F}_{256} \times \mathbb{F}_{256}$ such that $e = e_0 + e_1 \cdot \nu$ for ν a primitive element of $\mathbb{F}_{2^{16}}$ over $\mathbb{F}_{256}$ (i.e. $\mathbb{F}_{2^{16}} = \mathbb{F}_{256}[\nu]$). Such an element of $\mathbb{F}_{2^{16}}$ is serialized as $\mathsf{byte}(e_0) \parallel \mathsf{byte}(e_1)$. A vector $(v_1, \ldots, v_\ell) \in \mathbb{F}_{2^{16}}^\ell$ is hence naturally serialized as:

$$\mathsf{Serialize}(v) = \mathsf{byte}(v_{1,0}) \parallel \mathsf{byte}(v_{1,1}) \parallel \ldots \parallel \mathsf{byte}(v_{\ell,0}) \parallel \mathsf{byte}(v_{\ell,1})$$

where $v_i = v_{i,0} + v_{i,1} \cdot \nu$ for all $i \in [1, \ell]$.

The concrete values of ξ and ν which define the representation of $\mathbb{F}_{256}$ and $\mathbb{F}_{2^{16}}$ are provided in Section 3.

Tuples and arrays. MQOM further manipulates tuples of elements that might be of different natures. Such a tuple is naturally serialized as:

$$\mathsf{Serialize}((e_1, \ldots, e_\ell)) = \mathsf{Serialize}(e_1) \parallel \ldots \parallel \mathsf{Serialize}(e_\ell) \ .$$

In the same way, an array $\texttt{arr} = (\texttt{arr[0]}, \ldots, \texttt{arr}[\ell - 1])$ is serialized as

$$\mathsf{Serialize}(\texttt{arr}) = \mathsf{Serialize}(\texttt{arr[0]}) \parallel \ldots \parallel \mathsf{Serialize}(\texttt{arr}[\ell - 1]) \ .$$

2.4 Main algorithms

2.4.1 Key generation

The key generation of MQOM consists in pseudorandomly generating an MQ instance, with triangular matrices A_i. It randomly draws a master seed `seed_key` from which it derives the secret MQ solution x and another seed `mseed_eq` from which the MQ equations $\{A_i\}, \{b_i\}$ are derived. The MQ output y is then computed from $\{A_i\}, \{b_i\}$ and x. Finally, the key pair is defined and returned as $\mathtt{pk} := (\mathtt{mseed\_eq}, y)$ and $\mathtt{sk} := (\mathtt{pk}, x)$. The key generation is depicted in Algorithm 1. The subroutine ExpandEquations is invoked to expand the MQ equations from the seed `mseed_eq`. Subseeds `seed_eq[0]`, ..., `seed_eq[`$m-1$`]` are first derived from `mseed_eq`, then (A_i, b_i) is expanded from `seed_eq[`i`]` for every $i \in [1, m]$.

Algorithm 1 KeyGen()

Output: a secret key `sk`, a public key `pk`

1: $\mathtt{seed\_key} \leftarrow \{0,1\}^{2\lambda}$
2: $(x, \mathtt{mseed\_eq}) \leftarrow \mathsf{Parse}(\mathsf{XOF}_0(\mathtt{seed\_key}, \mathtt{len} := |x|_2 + 2\lambda))$ ▷ $x \in \mathbb{F}^n, \mathtt{mseed\_eq} \in \{0,1\}^{2\lambda}$
3: $(\{A_i\}, \{b_i\}) \leftarrow \mathsf{ExpandEquations}(\mathtt{mseed\_eq})$ ▷ $A_i \in \mathbb{F}^{n \times n}, b_i \in \mathbb{F}^n$
4: **for** $i = 0$ **to** $m - 1$ **do**
5: $\quad y_i \leftarrow x^\intercal A_i x + b_i^\intercal x$ ▷ $y_i \in \mathbb{F}$
6: $y \leftarrow (y_1, \ldots, y_m)$ ▷ $y \in \mathbb{F}^m$
7: $\mathtt{pk} \leftarrow \mathsf{Serialize}(\mathtt{mseed\_eq}, y)$
8: $\mathtt{sk} \leftarrow \mathsf{Serialize}(\mathtt{pk}, x)$
9: **return** $(\mathtt{pk}, \mathtt{sk})$

Algorithm 2 ExpandEquations(`mseed_eq`)

Input: a seed key $\mathtt{mseed\_eq} \in \{0,1\}^{2\lambda}$
Output: MQ equations $(\{A_i\}, \{b_i\})$

Let $nc_j = j \cdot \log_2 |\mathbb{F}|, \forall j \in [1, m]$ ▷ Number of PRG bits in jth row of A_i
Let $nz_j = (n - j) \cdot \log_2 |\mathbb{F}|, \forall j \in [1, m]$ ▷ Number of zero bits in jth row of A_i
Let $nb_j = \lceil nc_j / 8 \rceil, \forall j \in [1, m]$ ▷ Number of PRG bytes for the jth row of A_i
Let $nb_{\text{eq}} = nb_{n-1} + \sum_{j=0}^{n-1} nb_j$ ▷ Number of PRG bytes for (A_i, b_i)
for $i = 1$ **to** m **do**
$\quad \mathtt{seed\_eq}[i-1] \leftarrow \mathsf{XOF}_1((\mathtt{mseed\_eq}, \mathsf{Bits}_{16}(i)), \mathtt{len} := \lambda)$
$\quad (\mathtt{byte}[0] \parallel \ldots \parallel \mathtt{byte}[nb_{\text{eq}} - 1]) \leftarrow \mathsf{PRG}(0^\lambda, 0, \mathtt{seed\_eq}[i-1], nb_{\text{eq}})$
$\quad k \leftarrow 0$
$\quad$ **for** $j = 1$ **to** n **do**
$\quad\quad \mathtt{row\_bits} = \mathtt{bytes}[k] \parallel \cdots \parallel \mathtt{bytes}[k + nb_j - 1]$
$\quad\quad A_{i,j} \leftarrow \mathsf{Parse}(\mathsf{Truncate}_{nc_j}(\mathtt{row\_bits}) \parallel 0^{nz_j})$ ▷ $A_{i,j} \in \mathbb{F}^n$, jth row of A_i
$\quad\quad k \leftarrow k + nb_j$
$\quad A_i \leftarrow (A_{i,1}, \ldots, A_{i,n})$ ▷ $A_i \in \mathbb{F}^{n \times n}$
$\quad b_i \leftarrow \mathsf{Parse}(\mathtt{byte}[k] \parallel \cdots \parallel \mathtt{byte}[nb_{\text{eq}} - 1])$ ▷ $b_i \in \mathbb{F}^n$
return $(\{A_i\}, \{b_i\})$

2.4.2 Signing

The MQOM signing process is depicted in Algorithm 3 while the challenge sampling subroutine (implementing the grinding tweak) is depicted in Algorithm 4. The subroutines for the BLC commitment and computation of P_α are depicted in Section 2.5.

Algorithm 3 Sign($\mathtt{sk}, \mathtt{msg}$)

Input: a secret key $\mathtt{sk}$, a message $\mathtt{msg}$
Output: a signature sig

$(\mathtt{pk}, x) = \mathsf{Parse}(\mathtt{sk})$
$(\mathtt{mseed\_eq}, y) = \mathsf{Parse}(\mathtt{pk})$
$(\{A_i\}, \{b_i\}) \leftarrow \mathsf{ExpandEquations}(\mathtt{mseed\_eq})$
$\mathtt{mseed} \leftarrow \{0,1\}^\lambda$
$\mathtt{salt} \leftarrow \{0,1\}^\lambda$
$\mathtt{msg\_hash} \leftarrow \mathsf{Hash}_2(\mathtt{msg})$
$(\mathtt{com}_1, \mathtt{key}, x_0, u_0, u_1) \leftarrow \mathsf{BLC.Commit}(\mathtt{mseed}, \mathtt{salt}, x)$
$(\alpha_0, \alpha_1) \leftarrow \mathsf{ComputePAlpha}(\mathtt{com}_1, x_0, u_0, u_1, x, \{A_i\}, \{b_i\}, \{y_i\})$
$\mathtt{com}_2 \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
$\mathtt{hash} \leftarrow \mathsf{Hash}_4(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2, \mathtt{msg\_hash})$
$(i^*, \mathtt{nonce}) \leftarrow \mathsf{SampleChallenge}(\mathtt{hash})$
$\mathtt{opening} \leftarrow \mathsf{BLC.Open}(\mathtt{key}, i^*)$
return sig $:= \mathsf{Serialize}(\mathtt{salt}, \mathtt{com}_1, \mathtt{com}_2, \alpha_1, \mathtt{opening}, \mathtt{nonce})$

Algorithm 4 SampleChallenge($\mathtt{hash}$)

Input: Fiat-Shamir hash digest $\mathtt{hash} \in \{0,1\}^{2\lambda}$
Output: challenge indexes $i^* \in [0, N-1]^\tau$, a grinding counter $\mathtt{nonce} \in [0, 2^{32}-1]$

$\mathtt{nonce} \leftarrow 0$ ▷ $\mathtt{nonce} \in [0, 2^{32}-1]$
$(i^*, \mathtt{val}) \leftarrow \mathsf{Parse}(\mathsf{XOF}_5((\mathtt{hash}, \mathtt{nonce}), \mathtt{len} := \tau \cdot \log_2(N) + w))$ ▷ $i^* \in [0, N-1]^\tau$
while $\mathtt{val} \neq 0$ **do** ▷ $\mathtt{val} \in [0, 2^w - 1]$
    $\mathtt{nonce} \leftarrow \mathtt{nonce} + 1$
    $(i^*, \mathtt{val}) \leftarrow \mathsf{Parse}(\mathsf{XOF}_5((\mathtt{hash}, \mathtt{nonce}), \mathtt{len} := \tau \cdot \log_2(N) + w))$
return $(i^*, \mathtt{nonce})$

2.4.3 Verification

The MQOM verification process is depicted in Algorithm 5. The subroutines for the BLC evaluation and recomputation of P_α are depicted in Section 2.5.

Algorithm 5 Verif($\mathtt{pk}, \mathtt{msg}, \mathsf{sig}$)

Input: a public key $\mathtt{pk}$, a message $\mathtt{msg}$, a signature sig
Output: True or False

1: $(\mathtt{mseed\_eq}, y) = \mathsf{Parse}(\mathtt{pk})$
2: $(\{A_i\}, \{b_i\}) \leftarrow \mathsf{ExpandEquations}(\mathtt{mseed\_eq})$
3: $(\mathtt{salt}, \mathtt{com}_1, \mathtt{com}_2, \alpha_1, \mathtt{opening}, \mathtt{nonce}) = \mathsf{Parse}(\mathsf{sig})$
4: $\mathtt{msg\_hash} \leftarrow \mathsf{Hash}_2(\mathtt{msg})$
5: $\mathtt{hash} \leftarrow \mathsf{Hash}_4(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2, \mathtt{msg\_hash})$
6: $(i^*, \mathtt{val}) \leftarrow \mathsf{Parse}(\mathsf{XOF}_5((\mathtt{hash}, \mathtt{nonce}), \mathtt{len} := \tau \cdot \log_2(N) + w))$
7: **if** $\mathtt{val} \neq 0$ **then return** False
8: $(\mathtt{ret}, \mathtt{x\_eval}, \mathtt{u\_eval}) \leftarrow \mathsf{BLC.Eval}(\mathtt{salt}, \mathtt{com}_1, \mathtt{opening}, i^*)$
9: **if** $\mathtt{ret} \neq$ True **then return** False
10: $\alpha_0 \leftarrow \mathsf{RecomputePAlpha}(\mathtt{com}_1, \alpha_1, \mathtt{x\_eval}, \mathtt{u\_eval}, \{A_i\}, \{b_i\}, \{y_i\})$
11: $\mathtt{com}_2' \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
12: **if** $\mathtt{com}_2' \neq \mathtt{com}_2$ **then return** False
13: **return** True

2.5 Subroutines

2.5.1 Arithmetic routines

The main arithmetic routine of the signing process is ComputePAlpha which computes the polynomials $P_\alpha = P_u + \Phi(F(P_x))$, for all the executions $e \in [0, \tau - 1]$, where $F = (f_1, \ldots, f_m)$ with $f_i(x) = x^\intercal A_i x + b_i^\intercal x - y_i$. The resulting polynomials are returned as arrays of coefficient vectors: $\alpha_0 = (\alpha_0[0], \ldots, \alpha_0[\tau - 1])$ and $\alpha_1 = (\alpha_1[0], \ldots, \alpha_1[\tau - 1])$ such that for a given execution e, the polynomial P_α is defined as: $P_\alpha = \alpha_0[e] + \alpha_1[e] \cdot X$. This function relies on the subroutine ComputePz which computes $P_z = F(P_x)$ from P_x. The batching (a.k.a. 5-round) variant is enabled/disabled with the Boolean `batching`.

Algorithm 6 ComputePAlpha$(\texttt{com}, x_0, u_0, u_1, x, \{A_i\}, \{b_i\})$

Input: a BLC commitment `com`, coefficient arrays x_0, u_0, u_1, MQ secret solution x, MQ equations $\{A_i\}, \{b_i\}$
Output: coefficient arrays α_0, α_1

if batching **then** ▷ batching = True ⇒ 5-round variant
  $\Gamma \in \mathbb{K}^{\eta \times \frac{m}{\mu}} \leftarrow$ Parse(XOF$_8$(`com`, `len` $:= \eta \cdot m \cdot \log_2 |\mathbb{F}|$))
else ▷ batching = False ⇒ 3-round variant
  $\Gamma \leftarrow I_\eta$ ▷ Identity matrix $I_\eta \in \mathbb{K}^{\eta \times \eta}$ with $\eta = m/\mu$
for $e = 0$ **to** $\tau - 1$ **do**
  $(z_0, z_1) \leftarrow$ ComputePz$(x_0[e], x, \{A_i\}, \{b_i\})$ ▷ $P_z = z_0 + z_1 \cdot X \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^m$
  $\alpha_0[e] \leftarrow u_0[e] + \Gamma \cdot \Phi(z_0)$
  $\alpha_1[e] \leftarrow u_1[e] + \Gamma \cdot \Phi(z_1)$ ▷ $P_\alpha = \alpha_0[e] + \alpha_1[e] \cdot X \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^\eta$
return (α_0, α_1)

Algorithm 7 ComputePz$(x_0[e], x, \{A_i\}, \{b_i\})$

Input: coefficient vectors $x_0[e] \in \mathbb{K}^n$, $x \in \mathbb{F}^n$, MQ equations $\{A_i\}, \{b_i\}$
Output: coefficients $z_0, z_1 \in \mathbb{K}^m$

▷ Compute $P_{z,i} = P_x{}^\intercal A_i P_x + b_i^\intercal P_x \cdot X - y_i \cdot X^2$ for all $i \in [1, m]$
▷ Skip computation of degree-2 coefficients (known to be 0)
for $i = 1$ **to** m **do**
  ▷ Compute $P_t = t_0 + t_1 \cdot X := A_i \cdot P_x + b_i \cdot X$
  $t_0 \leftarrow A_i \cdot x_0[e]$ ▷ $t_0 \in \mathbb{K}^n$
  $t_1 \leftarrow A_i \cdot x + b_i$ ▷ $t_1 \in \mathbb{F}^n$
  ▷ Compute $P_{z,i} = z_{0,i} + z_{1,i} \cdot X = P_t^\intercal P_x - y_i \cdot X^2$
  $z_{0,i} \leftarrow t_0^\intercal \cdot x_0[e]$ ▷ $z_{0,i} \in \mathbb{K}$
  $z_{1,i} \leftarrow t_0^\intercal \cdot x + t_1^\intercal \cdot x_0[e]$ ▷ $z_{1,i} \in \mathbb{K}$
$z_0 \leftarrow (z_{0,1}, \ldots, z_{0,m})$ ▷ $z_0 \in \mathbb{K}^m$
$z_1 \leftarrow (z_{1,1}, \ldots, z_{1,m})$ ▷ $z_1 \in \mathbb{K}^m$
return (z_0, z_1)

Remark 3. *In* ComputePz*, the variable* $t_1 = A_i \cdot x + b_i$ *takes a different value for each* i*, but for a fixed* i*, its value remains constant across all executions* $e = 0, \ldots, \tau - 1$*. This implies that the*

m values of t_1 (corresponding to $i \in [1, m]$) can be computed once and reused for all executions. Furthermore, since these values depend only on the secret key `sk`*, they could be precomputed and used for every call to the signing algorithm. We do not consider this optimization here to keep the description simple but it could be easily integrated to enhance the efficiency of a concrete implementation.*

The main arithmetic routine of the verification process is RecomputePAlpha which recomputes the polynomials P_α from the coefficients α_1 and the opened evaluations $P_x(r)$, $P_u(r)$ for all the executions $e \in [0, \tau - 1]$. Namely, this function recomputes and returns the missing coefficients α_1. It makes use of the subroutine ComputePzEval which computes the evaluation $P_z(r)$ from $P_x(r)$, $P_u(r)$ for a single execution.

Algorithm 8 RecomputePAlpha$(\texttt{com}, \alpha_1, i^*, \mathsf{x_eval}, \mathsf{u_eval}, \{A_i\}, \{b_i\}, \{y_i\})$

Input: a BLC commitment `com`, coefficient array α_1, index array i^*, evaluation arrays x_eval, u_eval, MQ equations $\{A_i\}, \{b_i\}, \{y_i\}$
Output: coefficient array α_0

if batching **then** ▷ batching = True ⇒ 5-round variant
  $\Gamma \in \mathbb{K}^{\eta \times \frac{m}{\mu}} \leftarrow$ Parse(XOF$_8$(`com`, `len` $:= \eta \cdot m \cdot \log_2 |\mathbb{F}|$))
else ▷ batching = False ⇒ 3-round variant
  $\Gamma \leftarrow I_\eta$ ▷ Identity matrix $I_\eta \in \mathbb{K}^{\eta \times \eta}$ with $\eta = m/\mu$
for $e = 0$ **to** $\tau - 1$ **do**
  Let $r = \omega_{i^*[e]}$ ▷ $r \in \mathbb{K}$
  Let $v_x = \mathsf{x\_eval}[e]$ ▷ $v_x = P_x(r) \in \mathbb{K}^n$
  Let $v_u = \mathsf{u\_eval}[e]$ ▷ $v_u = P_u(r) \in \mathbb{K}^\eta$
  $v_z \leftarrow$ ComputePzEval$(r, v_x, \{A_i\}, \{b_i\}, \{y_i\})$ ▷ $v_z = P_z(r) \in \mathbb{K}^\eta$
  $v_\alpha \leftarrow v_u + \Gamma \cdot \Phi(v_z)$ ▷ $v_\alpha = P_\alpha(r) = \alpha_0[e] + \alpha_1[e] \cdot r$
  $\alpha_0[e] \leftarrow v_\alpha - \alpha_1[e] \cdot r$ ▷ $\alpha_0[e] \in \mathbb{K}^\eta$
return α_0

Algorithm 9 ComputePzEval$(r, v_x, \{A_i\}, \{b_i\}, \{y_i\})$

Input: evaluation point $r \in \Omega$, evaluation $v_x \in \mathbb{K}$, MQ equations $\{A_i\}, \{b_i\}, \{y_i\}$
Output: evaluation $v_z \in \mathbb{K}$

▷ Compute $v_{z,i} = {v_x}^\intercal A_i v_x + b_i^\intercal v_x \cdot r - y_i \cdot r^2$ for all $i \in [1, m]$
for $i = 1$ **to** m **do**
  ▷ Compute $v_t = P_t(r) = A_i \cdot P_x(r) + b_i \cdot r$
  $v_t \leftarrow A_i \cdot v_x + b_i \cdot r$ ▷ $v_t \in \mathbb{K}^n$
  ▷ Compute $v_{z,i} = P_{z,i}(r) = v_t^\intercal v_x - y_i \cdot r^2$
  $v_{z,i} \leftarrow v_t^\intercal \cdot v_x - y_i \cdot r^2$ ▷ $v_{z,i} \in \mathbb{K}$
$v_z \leftarrow (v_{z,1}, \ldots, v_{z,m})$ ▷ $v_z \in \mathbb{K}^m$
return v_z

2.5.2 Batch line commitment routines

The BLC.Commit routine computes the BLC commitment $\texttt{com}_1$, the associated opening key (the nodes of the GGM trees), and the associated polynomials P_x, P_u returned as array of coefficients.

Algorithm 10 BLC.Commit($\texttt{mseed}, \texttt{salt}, x$)

Input: a master seed $\texttt{mseed} \in \{0,1\}^\lambda$, a salt $\texttt{salt} \in \{0,1\}^\lambda$, an MQ secret solution x
Output: a BLC commitment $\texttt{com}_1$, an opening key $\texttt{key}$, coefficient arrays x_0, u_0, u_1

$(\texttt{rseed}[0], \ldots, \texttt{rseed}[\tau-1]) \leftarrow \mathsf{Parse}(\mathsf{PRG}(0^\lambda, 0, \texttt{mseed}, \tau \cdot \lambda))$
$\delta \leftarrow \mathsf{FirstBits}_\lambda(x)$ ▷ $\delta \in \{0,1\}^\lambda$
for $e = 0$ **to** $\tau - 1$ **do**
  $(\texttt{node}[e], \texttt{lseed}[e]) \leftarrow \mathsf{GGMTree.Expand}(\texttt{salt}, \texttt{rseed}[e], e, \delta)$
  $\texttt{tweaked\_salt} \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 0, e, 0)$
  for $i = 0$ **to** $N - 1$ **do**
    $\texttt{ls\_com}[e][i] \leftarrow \mathsf{SeedCommit}(\texttt{tweaked\_salt}, \texttt{lseed}[e][i])$
    $\texttt{exp}[e][i] \leftarrow \mathsf{PRG}(\texttt{salt}, e, \texttt{lseed}[e][i], |x|_8 + |u|_8 - \lambda/8)$
    $(\bar{x}_i, \bar{u}_i) \leftarrow \mathsf{Parse}(\texttt{lseed}[e][i] \,\|\, \texttt{exp}[e][i])$ ▷ $\bar{x}_i \in \mathbb{F}^n$, $\bar{u}_i \in \mathbb{K}^\eta$
  $\texttt{hash\_ls\_com}[e] \leftarrow \mathsf{Hash}_6(\texttt{ls\_com}[e])$
  ▷ Compute $P_u = u_0[e] + u_1[e] \cdot X = \sum_{i=0}^{N-1} \bar{u}_i \cdot (X - \omega_i)$
  $u_0[e] \leftarrow -\sum_{i=0}^{N-1} \omega_i \cdot \bar{u}_i$ ▷ $u_0[e] \in \mathbb{K}^\eta$
  $u_1[e] \leftarrow \sum_{i=0}^{N-1} \bar{u}_i$ ▷ $u_1[e] \in \mathbb{K}^\eta$
  ▷ Compute $P_x = x_0[e] + x \cdot X = \Delta_x[e] \cdot X + \sum_{i=0}^{N-1} \bar{x}_i \cdot (X - \omega_i)$
  $x_0[e] \leftarrow -\sum_{i=0}^{N-1} \omega_i \cdot \bar{x}_i$ ▷ $x_0[e] \in \mathbb{K}^n$
  $\Delta_x[e] \leftarrow x - \sum_{i=0}^{N-1} \bar{x}_i$ ▷ $\Delta_x[e] \in \mathbb{F}^n$
  $\Delta_x^{(1)}[e] \leftarrow \mathsf{NextBits}_\lambda(\Delta_x[e])$ ▷ $\Delta_x^{(1)}[e] \in \{0,1\}^{|x|_2 - \lambda}$
$\texttt{com}_1 \leftarrow \mathsf{Hash}_7(\texttt{hash\_ls\_com}, \Delta_x^{(1)})$
$\texttt{key} \leftarrow \mathsf{Serialize}(\texttt{node}, \texttt{ls\_com}, \Delta_x^{(1)})$
return $(\texttt{com}_1, \texttt{key}, x_0, u_0, u_1)$

From an opening key and an index array i^*, the BLC.Open routine returns the opening tuple made of the sibling paths ($\texttt{path}$), the hidden leaf commitments ($\texttt{out\_ls\_com}$) and the correction values ($\Delta_x^{(1)}$).

Algorithm 11 BLC.Open($\texttt{key}, i^*$)

Input: a commitment key $\texttt{key}$, an index array i^*
Output: a BLC opening $\texttt{opening} \in \{0,1\}^{\lambda \cdot \tau \cdot (\log_2(N)+2)}$

$(\texttt{node}, \texttt{ls\_com}, \Delta_x^{(1)}) \leftarrow \mathsf{Parse}(\texttt{key})$
for $e = 0$ **to** $\tau - 1$ **do**
  $\texttt{path}[e] \leftarrow \mathsf{GGMTree.Open}(\texttt{node}[e], i^*[e])$
  $\texttt{out\_ls\_com}[e] \leftarrow \texttt{ls\_com}[e][i^*[e]]$
$\texttt{opening} \leftarrow \mathsf{Serialize}(\texttt{path}, \texttt{out\_ls\_com}, \Delta_x^{(1)})$
return $\texttt{opening}$

The BLC.Eval routine is called by the verification algorithm to check the opening validity and derive the underlying evaluations $P_x(r)$, $P_u(r)$ for all the executions $e \in [0, \tau - 1]$.

Algorithm 12 BLC.Eval($\texttt{salt}, \texttt{com}, \texttt{opening}, i^*$)

Input: a salt `salt`, a BLC commitment `com`, a BLC opening `opening`, index array i^*
Output: `ret` $\in \{\mathsf{True}, \mathsf{False}\}$, evaluation arrays `x_eval`, `u_eval`

$(\texttt{path}, \texttt{out\_ls\_com}, \Delta_x^{(1)}) \leftarrow \mathsf{Parse}(\texttt{opening})$
for $e = 0$ **to** $\tau - 1$ **do**
  ▷ Compute partial GGM trees
  $\texttt{lseed}[e] \leftarrow \mathsf{GGMTree.PartiallyExpand}(\texttt{salt}, \texttt{path}[e], i^*[e])$
  ▷ Compute evaluations
  $\texttt{tweaked\_salt} \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 0, e, 0)$
  for $i = 0$ **to** $N - 1$ **do**
    if $i \neq i^*[e]$ **then**
      $\texttt{ls\_com}[e][i] \leftarrow \mathsf{SeedCommit}(\texttt{tweaked\_salt}, \texttt{lseed}[e][i])$
      $\texttt{exp}[e][i] \leftarrow \mathsf{PRG}(\texttt{salt}, e, \texttt{lseed}[e][i], |x|_8 + |u|_8 - \lambda/8)$
      $(\bar{x}_i, \bar{u}_i) \leftarrow \mathsf{Parse}(\texttt{lseed}[e][i] \parallel \texttt{exp}[e][i])$ ▷ $\bar{x}_i \in \mathbb{F}^n$, $\bar{u}_i \in \mathbb{K}^\eta$
  $\texttt{ls\_com}[e][i^*[e]] \leftarrow \texttt{out\_ls\_com}[e]$
  $\texttt{hash\_ls\_com}[e] \leftarrow \mathsf{Hash}_6(\texttt{ls\_com}[e])$
  Let $r = \omega_{i^*[e]}$ ▷ $r \in \mathbb{K}$
  $\Delta_x[e] \leftarrow \mathsf{PadLeft}_\lambda(\Delta_x^{(1)}[e])$ ▷ $\Delta_x[e] \in \mathbb{F}^n$
  $v_x \leftarrow \Delta_x[e] \cdot r + \sum_{i \neq i^*[e]} \bar{x}_i \cdot (r - \omega_i)$ ▷ $v_x = P_x(r) \in \mathbb{K}^n$
  $v_u \leftarrow \sum_{i \neq i^*[e]} \bar{u}_i \cdot (r - \omega_i)$ ▷ $v_u = P_u(r) \in \mathbb{K}^\eta$
  $\texttt{x\_eval}[e] \leftarrow v_x$
  $\texttt{u\_eval}[e] \leftarrow v_u$
▷ *Verify opening*
$\texttt{com}' \leftarrow \mathsf{Hash}_7(\texttt{hash\_ls\_com}, \Delta_x^{(1)})$
if $\texttt{com}' \neq \texttt{com}$ **then**
  return $(\mathsf{False}, \perp, \perp)$
return $(\mathsf{True}, \texttt{x\_eval}, \texttt{u\_eval})$

2.5.3 GGM tree routines

The GGM tree subroutines are depicted hereafter. The routine GGMTree.Expand (which is called in BLC.Commit) expands a GGM tree from a root seed with a salt, an execution index e (for domain separation) and the offset δ (for correlated tree optimization). It returns the derived list of nodes and leaf seeds. The routine GGMTree.Open (which is called in BLC.Open) extracts the sibling path from the nodes for a given hidden index. The routine GGMTree.PartiallyExpand (which is called in BLC.Eval) expands a GGM tree partially from a sibling path with a salt, an execution index e (for domain separation) and the underlying hidden index. The reader is referred to Figure 3 for an illustration of the tree structure and numbering of nodes.

Algorithm 13 GGMTree.Expand($\mathtt{salt}, \mathtt{rseed}[e], e, \delta$)

Input: a salt $\mathtt{salt} \in \{0,1\}^\lambda$, a root seed $\mathtt{rseed}[e] \in \{0,1\}^\lambda$, an execution index $e \in [0, \tau-1]$, an offset $\delta \in \{0,1\}^\lambda$
Output: a tree node array $\mathtt{node}[e]$, a leaf seed array $\mathtt{lseed}[e]$

$\mathtt{node}[e][2] \leftarrow \mathtt{rseed}[e]$
$\mathtt{node}[e][3] \leftarrow \mathtt{rseed}[e] \oplus \delta$
for $j = 1$ **to** $\log_2(N) - 1$ **do**
  $\mathtt{tweaked\_salt} = \mathsf{TweakSalt}(\mathtt{salt}, 2, e, j)$
  for $k = 2^j$ **to** $2^{j+1} - 1$ **do**
    $\mathtt{node}[e][2k] \leftarrow \mathsf{SeedDerive}(\mathtt{tweaked\_salt}, \mathtt{node}[e][k])$
    $\mathtt{node}[e][2k+1] \leftarrow \mathtt{node}[e][2k] \oplus \mathtt{node}[e][k]$
for $i = 0$ **to** $N - 1$ **do**
  $\mathtt{lseed}[e][i] \leftarrow \mathtt{node}[e][N+i]$
return $(\mathtt{node}[e], \mathtt{lseed}[e])$

Algorithm 14 GGMTree.Open($\mathtt{node}[e], i^*[e]$)

Input: a tree node array $\mathtt{node}[e]$, a hidden leaf index $i^*[e]$
Output: sibling path $\mathtt{path}[e]$

$i \leftarrow N + i^*[e]$ ▷ i: index of the hidden node at layer j
for $j = 0$ **to** $\log_2(N) - 1$ **do**
  $\mathtt{path}[e][j] \leftarrow \mathtt{node}[e][i \oplus 1]$
  $i \leftarrow \lfloor i/2 \rfloor$
return path

Algorithm 15 GGMTree.PartiallyExpand($\mathtt{salt}, \mathtt{path}[e], e, i^*[e]$)

Input: a salt $\mathtt{salt} \in \{0,1\}^\lambda$, a sibling path $\mathtt{path}[e]$, an execution index $e \in [0, \tau-1]$, a hidden leaf index $i^*[e] \in [0, N-1]$
Output: a partial leaf seed array $\mathtt{lseed}[e]$

▷ Initialize nodes to $\perp$
$(\mathsf{nodes}[e][1], \ldots, \mathsf{nodes}[e][2N-1]) = (\perp, \ldots, \perp)$
▷ Assign nodes with sibling path
$i \leftarrow N + i^*[e]$ ▷ i: index of the hidden node at layer j
for $j = 0$ **to** $\log_2(N) - 1$ **do**
    $\mathtt{node}[e][i \oplus 1] \leftarrow \mathtt{path}[e][j]$
    $i \leftarrow \lfloor i/2 \rfloor$
▷ Derive nodes from sibling path
for $j = 1$ **to** $\log_2(N) - 1$ **do**
    $\mathtt{tweaked\_salt} = \mathsf{TweakSalt}(\mathtt{salt}, 2, e, j)$
    for $k = 2^j$ **to** $2^{j+1} - 1$ **do**
        if $\mathtt{node}[e][k] \neq \perp$ **then**
            $\mathtt{node}[e][2k] \leftarrow \mathsf{SeedDerive}(\mathtt{tweaked\_salt}, \mathtt{node}[e][k])$
            $\mathtt{node}[e][2k+1] \leftarrow \mathtt{node}[e][2k] \oplus \mathtt{node}[e][k]$
for $i = 0$ **to** $N - 1$ **do**
    $\mathtt{lseed}[e][i] \leftarrow \mathtt{node}[e][N+i]$
return $\mathtt{lseed}[e]$

2.5.4 Seed processing routines

The following algorithms depict the subroutines of the GGM trees that are used to derive, commit and expand the seeds.

Algorithm 16 TweakSalt($\texttt{salt}, \texttt{sel}, e, j$)

Input: a salt $\texttt{salt} \in \{0,1\}^\lambda$, a selector $\texttt{sel} \in \{0,1,2\}$, an execution index $e \in [0, \tau - 1]$, a tree layer $j \in [0, \log_2(N) - 1]$
Output: a tweaked salt $\texttt{tweaked\_salt} \in \{0,1\}^\lambda$

▷ $\texttt{sel} = 0$ for seed commitment (first part)
▷ $\texttt{sel} = 1$ for seed commitment (second part)
▷ $\texttt{sel} = 2$ for seed derivation
▷ $\texttt{sel} = 3$ for PRG

1: $\texttt{tweak} \leftarrow \texttt{sel} + 4 \cdot e + 256 \cdot j$
2: $\texttt{tweaked\_salt} \leftarrow \texttt{salt} \oplus \mathsf{Bits}_\lambda(\texttt{tweak})$
3: **return** $\texttt{tweaked\_salt}$

Algorithm 17 SeedDerive($\texttt{tweaked\_salt}, \mathsf{seed}$)

Input: a tweaked salt $\texttt{tweaked\_salt} \in \{0,1\}^\lambda$, a seed $\mathsf{seed} \in \{0,1\}^\lambda$
Output: a derived seed $\texttt{new\_seed} \in \{0,1\}^\lambda$

1: $\texttt{new\_seed} \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt}, \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
2: **return** $\texttt{new\_seed}$

Algorithm 18 SeedCommit($\texttt{tweaked\_salt}, \mathsf{seed}$)

Input: a tweaked salt $\texttt{tweaked\_salt} \in \{0,1\}^\lambda$, a seed $\mathsf{seed} \in \{0,1\}^\lambda$
Output: a seed commitment $\texttt{seed\_com} \in \{0,1\}^{2\lambda}$

1: $\texttt{com}_1 \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt}, \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
2: $\texttt{com}_2 \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt} \oplus \mathsf{Bits}_\lambda(1), \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
3: $\texttt{seed\_com} \leftarrow \texttt{com}_1 \,\|\, \texttt{com}_2$
4: **return** $\texttt{seed\_com}$

Algorithm 19 PRG($\texttt{salt}, e, \mathsf{seed}, n_{\text{bytes}}$)

Input: a seed $\mathsf{seed} \in \{0,1\}^\lambda$, an execution index $e \in [0, \tau - 1]$, a salt $\texttt{salt} \in \{0,1\}^\lambda$, a number of bytes $n_{\text{bytes}} \in \mathbb{N}$
Output: a pseudorandom byte string $\texttt{out} \in \{0,1\}^{8 \cdot n_{\text{bytes}}}$

1: $n_{\text{blocks}} \leftarrow \lceil 8 \cdot n_{\text{bytes}} / \lambda \rceil$
2: **for** $i = 0$ **to** $n_{\text{blocks}} - 1$ **do**
3: $\quad \texttt{tweaked\_salt} \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 3, e, i)$
4: $\quad \texttt{block}[i] \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt}, \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
5: $(\texttt{byte}[0] \,\|\, \ldots \,\|\, \texttt{byte}[n_{\text{blocks}} \cdot \lambda/8 - 1]) \leftarrow \mathsf{Parse}(\texttt{block}[0] \,\|\, \ldots \,\|\, \texttt{block}[n_{\text{blocks}} - 1])$
6: $\texttt{out} \leftarrow \texttt{byte}[0] \,\|\, \ldots \,\|\, \texttt{byte}[n_{\text{bytes}} - 1]$
7: **return** $\texttt{out}$

2.5.5 Symmetric primitives

MQOM relies on two symmetric primitives:

1. A block cipher:
$$\mathsf{Enc} : (\mathtt{key}, \mathtt{ptx}) \in \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \mathtt{ctx} \in \{0,1\}^\lambda \ ;$$
2. An extendable-output hash function:
$$\mathsf{XOF} : (\mathtt{in}, \mathtt{len}) \in \{0,1\}^* \times \mathbb{N} \mapsto \mathtt{out} \in \{0,1\}^{\mathtt{len}} \ .$$

We further define a (fixed-length output) hash function as:
$$\mathsf{Hash} : \mathtt{in} \in \{0,1\}^* \mapsto \mathsf{XOF}(\mathtt{in}, 2\lambda) \in \{0,1\}^{2\lambda} \ .$$

Table 4 summarizes the instantiations of these two primitives for the NIST Categories I, III and V (corresponding to a parameter $\lambda = 128$, $\lambda = 192$, $\lambda = 256$ respectively). For Category III ($\lambda = 192$), Enc is defined as a truncated version of Rijndael-256-256 (hence the asterisk), which is formally defined as:
$$\mathsf{Enc}^{(192)} : (\mathtt{key}, \mathtt{ptx}) \in \{0,1\}^{192} \times \{0,1\}^{192} \mapsto \mathsf{Truncate}_{192}(\mathsf{Enc}^{(256)}(\mathtt{key} \parallel 0^{64}, \mathtt{ptx} \parallel 0^{64})) \ .$$

Table 4: Symmetric primitives in MQOM.

	Category I ($\lambda = 128$)	Category III ($\lambda = 192$)	Category V ($\lambda = 256$)
Enc	AES-128	Rijndael-256-256*	Rijndael-256-256
XOF	SHAKE-128	SHAKE-256	SHAKE-256

Domain separation. We enforce domain separation for different calls to the XOF (or Hash) functions by prepending a byte representing the call index i to the data being hashed. Specifically, for $i \in \mathbb{N}$, with $i < 256$, we define:
$$\mathsf{XOF}_i(\mathtt{in}, \mathtt{len}) := \mathsf{XOF}(\mathsf{Bits}_8(i) \parallel \mathtt{in}, \mathtt{len}) \ ,$$
and consequently $\mathsf{Hash}_i(\mathtt{in}) = \mathsf{Hash}(\mathsf{Bits}_8(i) \parallel \mathtt{in})$.

Here is a summary of the invocations to the (extendable output) hash function with associated purposes:

- XOF_0: expansion of `seed_key` (secret key),
- XOF_1: expansion of `seed_eq` (MQ equations),
- Hash_2: message hash,
- Hash_3: hash commitment of α_0, α_1,
- Hash_4: Fiat-Shamir hash,
- XOF_5: challenge sampling (with grinding),
- Hash_6: hash commitment of leaf seed commitments,
- Hash_7: BLC commitment,
- XOF_8: generation of Γ (batching variant).

2.5.6 Bit manipulation

We define hereafter the bit manipulation functions. The function

$$\mathsf{Bits}_\ell : [0, 2^\ell - 1] \to \{0,1\}^\ell$$

takes as input an integer and returns its binary representation. The functions $\mathsf{FirstBits}_\lambda$ and $\mathsf{NextBits}_\lambda$ provide the λ first bits and $|x|_2 - \lambda$ next bits of a $|x|_2$-bit string. Formally, we define:

$$\mathsf{FirstBits}_\lambda : \Delta_x \in \mathbb{F}^n \mapsto \Delta_x^{(0)} \in \{0,1\}^\lambda$$

and

$$\mathsf{NextBits}_\lambda : \Delta_x \in \mathbb{F}^n \mapsto \Delta_x^{(1)} \in \{0,1\}^{|x|_2 - \lambda} \ .$$

where

$$(\Delta_x^{(0)} \parallel \Delta_x^{(1)}) = \mathsf{Serialize}(\Delta_x) \in \{0,1\}^{|x|_2} \ .$$

We further define the function $\mathsf{PadLeft}_\lambda$ as

$$\mathsf{PadLeft}_\lambda : \Delta_x^{(1)} \in \{0,1\}^{|x|_2 - \lambda} \mapsto \mathsf{Parse}(0^\lambda \parallel \Delta_x^{(1)}) \in \mathbb{F}^n \ .$$

Finally, the function

$$\mathsf{Truncate}_\ell : \{0,1\}^* \to \{0,1\}^\ell$$

returns the ℓ first bits of its input bit string.

3 MQOM instances

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 (in terms of signature size and running times).

3.1 Parameter selection

MQ parameters. Instead of considering prime field as in the first version, MQOM v2 relies on the binary fields. The main motivation for this update is to avoid rejection sampling and arithmetic-Boolean conversions. As first option, we chose $|\mathbb{F}| = 2$ for the base field, which leads to the shortest signatures. As second option, we chose $|\mathbb{F}| = 256$ which enjoys easier implementation (with a field element matching a byte). For those two fields, we took the number of equations m to be equal the number of unknowns n and selected the minimal $m = n$ to achieve a target security level (for categories I, III and V) according to the state of the art of MQ cryptanalysis (see Section 4.2).

Looking ahead, the extension field $\mathbb{K}$ is either defined as $\mathbb{F}_{256}$ or $\mathbb{F}_{2^{16}}$ (see explanations below). In order to ensure that m is always divisible by $\mu = [\mathbb{K} : \mathbb{F}]$ (which simplify the field-embedding batching – see Section 2.1.2), we restricted the selection to values of m that are multiples of 16 for $\mathbb{F} = \mathbb{F}_2$ and multiples of 2 for $\mathbb{F} = \mathbb{F}_{256}$.

Proof system parameters. The MQOM proof system relies on the parameters summarized in Table 2: the size N of the evaluation set $\Omega := \{\omega_0, \ldots, \omega_{N-1}\}$, the extension field $\mathbb{K}$ of degree μ, the number η of internal repetitions, the number τ of external repetitions and the grinding proof-of-work parameter w.

We chose N as a power of two to manipulate complete binary GGM trees. A larger N leads to a shorter signature at the cost of slower signing and verification algorithms. We chose to consider two values for N, namely $N = 2048$ (short variant) and $N = 256$ (fast variant), to obtain two different trade-offs between communication and computation. Then, the extension field $\mathbb{K}$ is chosen such that $|\mathbb{K}| \geq N$. To ease the implementation, we chose to consider a common $\mathbb{K}$ for the two base fields ($\mathbb{F}_2$ and $\mathbb{F}_{256}$) and thus define $\mathbb{K}$ as their common extension such that $|\mathbb{K}| \leq N$. This way, we get $\mathbb{K} = \mathbb{F}_{2^{16}}$ for $N = 2048$ and $\mathbb{K} = \mathbb{F}_{256}$ for $N = 256$.

Given the pair $(N, \mathbb{K})$, we selected the remaining parameters to achieve λ bits of soundness, with λ equal to 128, 192 and 256 for Categories I, III and V, respectively. On the one hand, we took $\eta = \lambda / \log_2 |\mathbb{K}|$ for the batching (5-round) variant to ensure a soundness error of $1/|\mathbb{K}|^\eta = 2^{-\lambda}$ (while η is fixed to as m/μ by design for the 3-round variant). On the other hand, we chose the parameters τ and w such that $\left(\frac{2}{N}\right)^\tau \cdot 2^{-w} \leq 2^{-\lambda}$ (see Section 2.1.4).

Field extension. As explained previously, the MQOM signature scheme relies on three different fields for $\mathbb{F}$ and $\mathbb{K}$: $\mathbb{F}_2$, $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$. Table 5 summarizes the field extensions that we use in our instances.

Table 5: Definition of field extensions.

Field ($\mathbb{F}$ or $\mathbb{K}$)	Field extension
$\mathbb{F}_{2^8}$	$\mathbb{F}_2[\xi]/\langle \xi^8 + \xi^4 + \xi^3 + \xi + 1 \rangle$
$\mathbb{F}_{2^{16}}$	$\mathbb{F}_{2^8}[\nu]/\langle \nu^2 + \nu + \xi^5 \rangle$

Evaluation domain. The PIOP evaluation query is sampled from the evaluation domain $\Omega := \{\omega_0, \ldots, \omega_{N-1}\}$ of size N. In our instances, we use

$$\omega_i := \nu \cdot \sum_{j=0}^{7}(b_{8+j} \cdot \xi^j) + \sum_{j=0}^{7}(b_j \cdot \xi^j) \in \mathbb{K}$$

for all $i \in \{0, N-1\}$, where $(b_0, \ldots, b_{15})$ is the binary decomposition of $i := \sum_{j=0}^{15} b_j \cdot 2^j$.

3.2 Key and signature sizes

Public key. The public key consists of a 2λ-bit seed `mseed_eq` for the generation of the MQ equations, and a serialized vector $y \in \mathbb{F}^m$ corresponding to the outputs of the equations. For $|\mathbb{F}| = 2$, we store 8 field elements on one byte. For $|\mathbb{F}| = 256$, we store one field element on one byte. Thus, the size of the public key is given by:

$$|\mathtt{pk}| = \begin{cases} \frac{2\lambda}{8} + \frac{m}{8} \text{ bytes} & \text{for } \mathbb{F}_2 \\ \frac{2\lambda}{8} + m \text{ bytes} & \text{for } \mathbb{F}_{256}. \end{cases}$$

(We recall that m is a multiple of 16 for $\mathbb{F} = \mathbb{F}_2$ so that $\frac{m}{8}$ is an integer.)

Secret key. The secret key consists of the same elements as the public key, plus a serialized vector $x \in \mathbb{F}_q^n$ corresponding to the secret solution of the MQ system. Thus, the size of the secret key is given by:

$$|\mathtt{sk}| = \begin{cases} \frac{2\lambda}{8} + \frac{m}{8} + \frac{n}{8} \text{ bytes} & \text{for } \mathbb{F}_2 \\ \frac{2\lambda}{8} + m + n \text{ bytes} & \text{for } \mathbb{F}_{256} \end{cases}$$

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 `seed_key`, the seed from which $(\mathtt{mseed\_eq}, y, x)$ are derived. In that case, the size of the secret key would be of $2\lambda/8$ bytes, but the signer would need to recompute `mseed_eq`, y and x at each signature, increasing the running time of the signing process. Moreover, the signing algorithm would be more sensitive to side-channel attacks. We hence recommend to use this alternative only if the size of the secret key is critical.

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

$\lvert\sigma\rvert = 32$	size of `nonce`.
$+ \lambda$	size of the `salt`
$+ 4\lambda$	size of $\mathtt{com}_1$ and $\mathtt{com}_2$
$+ \tau \cdot (\eta \cdot \mu \cdot \log_2 \lvert\mathbb{F}\rvert)$	size of α_1
$+ \tau \cdot (n \cdot \log_2 \lvert\mathbb{F}\rvert - \lambda)$	size of $\Delta_{x'}[e]$ in `opening`
$+ \tau \cdot \lambda \cdot \log_2 N$	size of `path` in `opening`
$+ \tau \cdot 2\lambda$	size of `out_ls_com` in `opening`

Given our encoding on field elements, $\log_2 |\mathbb{F}|$ should be replaced by 1 for $\mathbb{F}_2$ and by 8 for $\mathbb{F}_{256}$. We obtain the following sizes in bytes:

$$|\sigma| = 4 + \frac{\tau \cdot (n + \eta \cdot \mu)}{8} + \frac{5\lambda + \tau \cdot \lambda \cdot (\log_2 N + 1)}{8} \quad \text{for } \mathbb{F}_2,$$

and

$$|\sigma| = 4 + \tau \cdot \left(n + \eta \cdot \mu\right) + \frac{5\lambda + \tau \cdot \lambda \cdot (\log_2 N + 1)}{8} \quad \text{for } \mathbb{F}_{256}.$$

The only difference in terms of signature size between the 3-round and the 5-round variants, comes from the parameter η, which is a bit larger in the 3-round variant ($\eta = m/\mu$).

3.3 Proposed instances

All the signature parameters are summarized in Table 6, while the corresponding key and signature sizes are given in Table 7.

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

Parameter Sets	**NIST Security**	**MQ Parameters**		**Proof System Parameters**				
		$\lvert\mathbb{F}\rvert$	$m = n$	τ	N	μ	η	w
MQOM2-L1-gf2-short-3r/5r	Cat. I	2	160	12	2048	16	10/8	8
MQOM2-L1-gf2-fast-3r/5r	Cat. I	2	160	17	256	8	20/16	9
MQOM2-L1-gf256-short-3r/5r	Cat. I	256	48	12	2048	2	24/8	8
MQOM2-L1-gf256-fast-3r/5r	Cat. I	256	48	17	256	1	48/16	9
MQOM2-L3-gf2-short-3r/5r	Cat. III	2	240	18	2048	16	15/12	12
MQOM2-L3-gf2-fast-3r/5r	Cat. III	2	240	27	256	8	30/24	3
MQOM2-L3-gf256-short-3r/5r	Cat. III	256	72	18	2048	2	36/12	12
MQOM2-L3-gf256-fast-3r/5r	Cat. III	256	72	27	256	1	72/24	3
MQOM2-L5-gf2-short-3r/5r	Cat. V	2	320	25	2048	16	20/16	6
MQOM2-L5-gf2-fast-3r/5r	Cat. V	2	320	36	256	8	40/32	4
MQOM2-L5-gf256-short-3r/5r	Cat. V	256	96	25	2048	2	48/16	6
MQOM2-L5-gf256-fast-3r/5r	Cat. V	256	96	36	256	1	96/32	4

Table 7: The key and signature sizes in bytes.

Parameter Set	**Sizes (in bytes)**		
	pk	sk	Sig.
MQOM2-L1-gf2-short-3r/5r	52	72	2 868 / 2 820
MQOM2-L1-gf256-short-3r/5r	80	128	3 540 / 3 156
MQOM2-L1-gf2-fast-3r/5r	52	72	3 212 / 3 144
MQOM2-L1-gf256-fast-3r/5r	80	128	4 164 / 3 620
MQOM2-L3-gf2-short-3r/5r	78	108	6 388 / 6 280
MQOM2-L3-gf256-short-3r/5r	120	192	7 900 / 7 036
MQOM2-L3-gf2-fast-3r/5r	78	108	7 576 / 7 414
MQOM2-L3-gf256-fast-3r/5r	120	192	9 844 / 8 548
MQOM2-L5-gf2-short-3r/5r	104	144	11 764 / 11 564
MQOM2-L5-gf256-short-3r/5r	160	256	14 564 / 12 964
MQOM2-L5-gf2-fast-3r/5r	104	144	13 412 / 13 124
MQOM2-L5-gf256-fast-3r/5r	160	256	17 444 / 15 140

3.4 Benchmarks

Benchmarks for

- an AVX2-optimized implementation on an AVX2 machine are given in Table 8.
- an GFNI-optimized implementation on an AVX-512+GFNI machine are given in Table 9. GFNI brings native $\mathbb{F}_{256}$ multiplication acceleration in the Rijndael field.

Table 8: Benchmark of **AVX2-optimized implementation** of the MQOM on an AVX2 machine. Timings were run on an AMD Ryzen 7 PRO 6850U.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM2-L1-gf2-short-3r	0.64	1.74M	9.84	26.52M	9.32	25.11M
MQOM2-L1-gf2-short-5r	0.78	2.09M	11.82	31.84M	11.16	30.08M
MQOM2-L1-gf2-fast-3r	0.70	1.90M	5.24	14.12M	4.34	11.69M
MQOM2-L1-gf2-fast-5r	0.72	1.93M	5.39	14.53M	4.43	11.93M
MQOM2-L1-gf256-short-3r	0.16	0.44M	8.14	21.93M	7.65	20.60M
MQOM2-L1-gf256-short-5r	0.16	0.44M	6.55	17.65M	6.11	16.45M
MQOM2-L1-gf256-fast-3r	0.16	0.43M	2.60	7.01M	1.88	5.07M
MQOM2-L1-gf256-fast-5r	0.16	0.42M	2.28	6.15M	1.59	4.28M
MQOM2-L3-gf2-short-3r	3.05	8.23M	42.64	114.90M	38.44	103.60M
MQOM2-L3-gf2-short-5r	2.91	7.84M	40.12	108.12M	36.87	99.36M
MQOM2-L3-gf2-fast-3r	2.91	7.85M	21.29	57.37M	19.00	51.19M
MQOM2-L3-gf2-fast-5r	2.90	7.81M	21.10	56.85M	18.70	50.39M
MQOM2-L3-gf256-short-3r	0.72	1.95M	31.12	83.85M	27.61	74.41M
MQOM2-L3-gf256-short-5r	0.71	1.91M	24.76	66.71M	21.33	57.49M
MQOM2-L3-gf256-fast-3r	0.82	2.22M	12.68	34.18M	8.95	24.11M
MQOM2-L3-gf256-fast-5r	0.72	1.93M	10.19	27.47M	6.94	18.71M
MQOM2-L5-gf2-short-3r	5.12	13.81M	103.50	278.92M	98.90	266.51M
MQOM2-L5-gf2-short-5r	5.60	15.09M	113.90	306.94M	107.27	289.07M
MQOM2-L5-gf2-fast-3r	5.09	13.71M	54.00	145.51M	48.23	129.97M
MQOM2-L5-gf2-fast-5r	5.32	14.33M	55.90	150.65M	50.46	135.98M
MQOM2-L5-gf256-short-3r	1.24	3.35M	50.10	135.01M	45.12	121.59M
MQOM2-L5-gf256-short-5r	1.37	3.69M	46.38	124.98M	40.72	109.73M
MQOM2-L5-gf256-fast-3r	1.18	3.18M	20.29	54.69M	13.44	36.22M
MQOM2-L5-gf256-fast-5r	1.21	3.27M	19.53	52.64M	12.50	33.69M

Table 9: Benchmark of **optimized implementation** of the MQOM on an AVX-512+GFNI machine. Timings were run on an AMD Ryzen Threadripper PRO 7995WX.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM2-L1-gf2-short-3r	0.62	1.55M	6.01	15.00M	5.54	13.82M
MQOM2-L1-gf2-short-5r	0.63	1.57M	6.30	15.71M	5.61	13.99M
MQOM2-L1-gf2-fast-3r	0.62	1.55M	3.26	8.15M	2.58	6.45M
MQOM2-L1-gf2-fast-5r	0.62	1.55M	3.29	8.22M	2.59	6.47M
MQOM2-L1-gf256-short-3r	0.12	0.31M	5.74	14.34M	5.63	14.06M
MQOM2-L1-gf256-short-5r	0.12	0.31M	4.37	10.92M	4.25	10.61M
MQOM2-L1-gf256-fast-3r	0.11	0.28M	1.24	3.08M	1.05	2.61M
MQOM2-L1-gf256-fast-5r	0.12	0.30M	1.09	2.71M	0.89	2.23M
MQOM2-L3-gf2-short-3r	2.30	5.74M	21.05	52.54M	19.08	47.62M
MQOM2-L3-gf2-short-5r	2.50	6.25M	21.75	54.30M	19.34	48.27M
MQOM2-L3-gf2-fast-3r	2.27	5.67M	11.27	28.12M	9.39	23.44M
MQOM2-L3-gf2-fast-5r	2.29	5.71M	11.37	28.38M	10.04	25.05M
MQOM2-L3-gf256-short-3r	0.46	1.14M	17.15	42.81M	15.99	39.93M
MQOM2-L3-gf256-short-5r	0.46	1.15M	13.80	34.44M	12.74	31.80M
MQOM2-L3-gf256-fast-3r	0.52	1.29M	4.33	10.80M	4.03	10.05M
MQOM2-L3-gf256-fast-5r	0.46	1.16M	3.25	8.10M	2.97	7.42M
MQOM2-L5-gf2-short-3r	3.62	9.04M	37.43	93.44M	34.72	86.68M
MQOM2-L5-gf2-short-5r	3.65	9.11M	38.48	96.04M	32.15	80.26M
MQOM2-L5-gf2-fast-3r	3.53	8.82M	23.89	59.64M	19.37	48.34M
MQOM2-L5-gf2-fast-5r	3.53	8.82M	23.60	58.91M	19.32	48.23M
MQOM2-L5-gf256-short-3r	0.81	2.03M	24.36	60.80M	23.72	59.21M
MQOM2-L5-gf256-short-5r	0.73	1.82M	17.27	43.12M	16.85	42.06M
MQOM2-L5-gf256-fast-3r	0.81	2.02M	6.15	15.36M	5.46	13.62M
MQOM2-L5-gf256-fast-5r	0.82	2.04M	5.27	13.15M	4.58	11.44M

4 Security

4.1 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 $(\texttt{msg}_1, \ldots, \texttt{msg}_r)$ that they can select at will. The goal of the adversary is to generate a pair $(\texttt{msg}, \sigma)$ such that `msg` is not one of requests to the signature oracle and such that σ is a valid signature of `msg` 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.
- **Random Oracle Modem (ROM).** Our security statement holds in the ROM where the (extendable-output) hash function XOF is modelled as a random oracle.
- **Ideal Cipher Model (ICM).** Our security statement holds in the ICM where the block cipher Enc is modelled as an ideal cipher.

Based on the ROM and the ICM, the EUF-CMA security of MQOM holds from the soundness and zero-knowledge properties of the underlying ZK-PoK (which are overviewed in Section 2.1). The formal EUF-CMA security proof of MQOM will be added to a future version of the specification. It will heavily rely on usual techniques for MPC-in-the-Head signature schemes with GGM trees such as, e.g., the security proof of MQOM v1 [FR23a; BFR24] with specificities related to correlated GGM trees as in [KLS24].

4.2 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.

The best algorithms to solve polynomial systems are quite different over $\mathbb{F}_2$ and over larger finite fields. While the global asymptotic complexity of most algorithms is well-understood, estimating the concrete number of operations that is required to invert a given quadratic function is more an art than a science.

Several software tools provide estimates of the number of operations required to execute polynomial system solving algorithms, notably the `MQEstimator` [BMS+22] that is available in the `CryptographicEstimators` software library [EVZ+24].

The estimates provided by such tools should always be taken with a grain of salt. When estimating the number of bit operations required to run the F_5 algorithm, we observed a noticeable difference between the values given by version 1.1.1 (released on September 5th, 2023) and version 1.2.0 (released on November 24th, 2023) of the `MQEstimator` library. This in turn modifies the cost estimates for the hybrid-F_5 algorithm. The results are shown in Table 10.

n	m	q	v1.1.1		v1.2.0	
			F_5	hybrid-F_5 (k)	F_5	hybrid-F_5 (k)
36	36	256	150.3	111.5 (2)	203.8	143.2 (4)

Table 10: Cost estimates for solving quadratic systems with different versions of the `CryptographicEstimators` library. v1.1.1 was released on September 5th, 2023 (git commit `17924f39`) and v1.2.0 was released on November 24th, 2023 (git commit `35bc27a1`). The "F_5" (resp. hybrid-F_5) columns shows the log in base 2 of the number of "bit operations" required by the corresponding algorithm. For hybrid-F_5, the optimal number of "guessed" variables is given in parentheses.

The two versions also differ in the number of variables to "guess" in the hybrid-F_5 algorithm. These numbers can be obtained by the following bit of code (adjust with the required sizes):

```
q = 256
n = 36
m = 36
from cryptographic_estimators import MQEstimator
MQEstimator.MQEstimator(n, m, q).f5.time_complexity()
e = MQEstimator.MQEstimator(n, m, q).hybrid_f5
e.time_complexity(), e.k()
```

We traced down the difference between the two versions to a change in the default value of the linear algebra constant, namely the value of ω such that matrix multiplications requires $\mathcal{O}(n^\omega)$ arithmetic operations. The best algorithms to solve polynomial systems heavily rely on either sparse or dense linear algebra with exponentially large matrices. The best known value of the linear algebra constant is $\omega = 2.3728596$ [AW21] but it is well-known that the corresponding algorithms are so impractical that they have never been implemented (they are "galactic"). The CryptographicEstimators library switched from using $w = 2$ by default in version `1.1.1` to using $w = 2.81$ in version `1.2.0`, which corresponds to the use of the Strassen algorithm. This modification alone may explain the observed differences in cost estimates between the two versions. We agree that the use of the Strassen algorith for dense linear algebra is practical: it is implemented in M4RI and M4RIE [AB24] for binary matrices and in FFLAS/FFPACK [DGP08] for matrices over larger finite fields.

In any case, we discuss below a number of shortcomings of the CryptographicEstimators library, and discuss our own estimates.

4.2.1 Tools and building blocks

Arithmetic over $\mathbb{F}_{256}$. We consider that addition over $\mathbb{F}_{256}$ costs 8 bit operations (to XOR the two operands). Bernstein proposed in 2000 an algorithm to multiply two degree-7 polynomials over $\mathbb{F}_2[X]$ in 100 bit operations. Once their degree-14 product has been computed, it must be reduced modulo the irreducible polynomial that defines the finite field. Take $F = X^8 + X^4 + X^3 + X + 1$ (as given in Table 5). The remainder of a degree-14 polynomial modulo F can be computed with 28 bit operations. Therefore we consider that multiplication requires 128 bit operations.

Monomials. It is well-known that there are $\binom{n+d-1}{d}$ monomials of degree exactly d in n variables, and that there are $\binom{n+d}{d}$ monomials of degree at most d.

The situation is slightly different in the binary case, where considering the effect of the so-called "field equations" $x_i^2 = x_i$ is beneficial. In this case, we mostly work in the Boolean algebra

$$\mathcal{R} = \mathbb{F}_2[x_1, \ldots, x_n] \,/\, \left\langle x_1^2 - x_1, \ldots, x_n^2 - x_n \right\rangle.$$

In the Boolean algebra, there are $\binom{n}{d}$ monomials of degree exactly d and there are $\sum_{i=0}^{d} \binom{n}{i}$ monomials of degree at most d. This last sum has no closed expression.

Macaulay matrices. Consider a sequence of quadratic polynomials $f_1, \ldots, f_m$ in $\mathbb{F}_q[x_1, \ldots, x_n]$. Denote by I the ideal they span. I can be seen as an infinite-dimensional vector space spanned by the mf_j, where m ranges across all possible monomials. Let I_d (resp. $I_{\leq d}$) denote the subspace formed by the mf_j where m ranges across all monomials of degree d (resp. at most d). In general, I_d is not equal to the set of all degree-d polynomials of I, because of potential *degree falls*: some low-degree polynomials in I can only be obtained by taking a high-degree polynomial combination of the f_j. Both sets are equal only when the f_j are a Gröbner basis of I, a fact that was noted long ago by Lazard [Laz83].

Polynomials of I with a special shape can often be found effectively by means of linear algebra, by searching inside $I_{\leq d}$ for a sufficiently large d. The degree-d *Macaulay matrix* of $f_1, \ldots, f_m$ is the matrix whose rows are the $m_i f_j$ with $\deg m_i \leq d-2$ and whose columns correspond to all possible monomials of degree at most d. It follows that the row span of the degree-d Macaulay matrix is exactly $I_{\leq d}$.

The degree-d Macaulay matrix of $f_1, \ldots, f_m$ has $\binom{n+d}{d}$ columns and $m\binom{n+d-2}{d-2}$ rows. In the Boolean case, it has $\sum_{i=0}^{d} \binom{n}{i}$ columns and $m \sum_{i=0}^{d-2} \binom{n}{i}$ rows.

Macaulay matrices are quite sparse, because each row has at most $(n+1)(n+2)/2$ non-zero coefficients. It is well-known that they are also rank-deficient: there are linear dependencies between rows, at least because of the trivial relations $f_i f_j = f_j f_i$ and also because of $f_i^2 = f$ in the binary case. However, under the assumption that the f_j form a (semi-)regular sequence, the rank of the degree-d Macaulay matrix can be determined precisely and it only depends on n and m; it does not depend from the actual coefficients of the f_j's. This assumption essentially means that the polynomials are not bound by "unexpected" algebraic relations. It is usually well-verified in practice on unstructured systems, and is therefore standard in all the cryptographic literature. The interested reader is referred to [Bar04; BFS15] for more details. We now assume that the the f_j's are semi-regular.

The *leading-degree block* of the degree-d Macaulay matrix is the submatrix restricted to the $m_i f_j$ where $\deg m_i = d-2$ (so that $m_i \cdot f_j$ has degree exactly d) and restricted to the columns describing degree-d monomials. In other terms, it contains the degree-d terms of degree-d polynomials.

Consider the series expansion:

$$\sum_j a_j z^j = (1-z^2)^m (1-z)^{-n}.$$

The smallest index j such that $a_j \leq 0$ is the *degree of regularity* of the semi-regular sequence. When d is strictly less than the degree of regularity, then the rank of the leading-degree block of the degree-d Macaulay matrix is $\binom{n+d-1}{d} - a_d$ (in the non-binary case). If d is equal or greater than the degree of regularity, then the rank is just $\binom{n+d-1}{d}$ (the number of columns). Instead of explicitly working with the series, which requires tools from computer algebra, we use a simple

recurrence relations between the ranks of Macaulay matrices. Define

$$\begin{aligned} R_{0,j}(n,m) &= 0 \\ R_{1,j}(n,m) &= 0 \\ R_{d,0}(n,m) &= 0 \\ R_{d,j+1}(n,m) &= R_{d,j}(n,m) + \max\left(0, \binom{n}{d-2} - R_{d-2,j}(n,m)\right) \\ R_d(n,m) &= R_{d,m}(n,m) \end{aligned}$$

It is shown in [Bar04, lemma 3.3.2] that $R_d(n,m)$ is the rank of the leading-degree block of the degree-d Macaulay matrix, under the assumption that the f_j are (semi-)regular and that d is less than their degree of regularity. The smallest d such that $R_d(n,m)$ is greater than or equal to $\binom{n+d-1}{d}$, which is the number of degree-d monomials in n variables, is the degree of regularity of the f_j. Because the leading-degree blocks are clearly linearly independent from each other, the rank of the degree-d Macaulay matrix is $\sum_{i=0}^{d} R_i(n,m)$.

Over the binary field, and taking into account the field equations, the same reasoning can be applied, but the series is different:

$$\sum_j a_j z^j = (1+z)^n (1+z^2)^{-m}$$

and the recurrence relation is also different:

$$R_{d,j+1}(n,m) = R_{d,j}(n,m) + \max\left(0, \binom{n}{d-2} - R_{d-2,j+1}(n,m)\right)$$

Solving sparse linear systems with the block Wiedemann algorithm. In order to solve $Ax = b$ with a sparse matrix A, the Block Wiedemann algorithm is usually the solution of choice. We refer the reader to [CCN+12; BGG+20] for details about the algorithm and practical results. It has two main parameters, the "blocking factors", that we denote by $\tilde{m}$ and $\tilde{n}$. Let N denote the size of the matrix and $|A|$ denote the number of non-zero entries in A.

The bulk of the workload consists in "matrix-vector products", that are in fact (sparse $N \times N$ matrix) $\times$ (dense vector) products. It follows that each matrix-vector product requires $2|A|$ field operations (half additions and half multiplications). The block Wiedemann algorithm has three phases:

BW1 Split in $\tilde{n}$ independent jobs. Each job does $(1/\tilde{n} + 1/\tilde{m})N$ iterations in sequence. Each iteration does a "sparse matrix-dense vector" as described above, followed by a (dense $\tilde{m} \times N$ matrix) $\times$ vector product that requires $2N\tilde{m}$ operations (half additions, half multiplications). In total, there are $(1 + \tilde{n}/\tilde{m})N$ iterations aggregated over the $\tilde{n}$ independent jobs. After each iteration, a dense vector of size $\tilde{m}$ must be stored persistently. The total output size of the $\tilde{n}$ independent jobs is therefore $(\tilde{m} + \tilde{n})N$ field elements. This phase requires $(1/\tilde{n} + 1/\tilde{m})N$ sequential steps, even if the matrix-vector product itself is perfectly parallel. The total number of arithmetic operations is $2(1 + \tilde{n}/\tilde{m})N(|A| + \tilde{m}N)$.

BW2 Its input consists in $(\tilde{m} + \tilde{n})N$ field elements, it has quasi-linear running time complexity, and parallelizes well. Its memory usage may not be negligible though.

BW3 Can be split in a very high number of independent jobs. Does $N/\tilde{n}$ "sparse matrix-dense vector" products in total.

Choosing the optimal values of $\tilde{n}$ and $\tilde{m}$ is somewhat of an art. It is usual to choose $\tilde{m}/\tilde{n} = 2$ or $\tilde{m}/\tilde{n} = 3$. This yields a total workload of $(1.5 + 1/n)N$ or $(1.333 + 1/\tilde{n})N$ iterations, respectively. Increasing $\tilde{n}$ reduces the total time spent in BW3, and if $\tilde{m}/\tilde{n}$ is fixed, it does not increase the running time of BW1. However, increasing $\tilde{n}$ will increase the running time and the memory footprint of BW2. Note that increasing $\tilde{n}$ has another practical advantage, by increasing the level of coarse-grained parallelism in BW1.

Solving dense linear systems with the Strassen algorithm. Naive cubic multiplication of an $m \times k$ matrix by a $k \times n$ matrix requires mnk additions and as many multiplications in the base field. The Strassen-Winograd algorithm multiples two 2×2 matrices using 7 multiplications and 15 additions. This leads to a $\mathcal{O}\left(N^{2.807}\right)$ algorithm to multiply $N \times N$ matrices. But what is the constant hidden in the big Oh? To obtain realistic values, we implemented a naive estimator.

Matrix multiplication. Strassen requires more arithmetic operations than the cubic algorithm when N is very small (typically less than 16). Therefore, we assume that matrix multiplication uses the cubic algorithm as soon as one of the matrix dimension is strictly less than 17.

When the matrix dimensions are not even, we ignore the last row/column and process it afterwards naively (with a matrix-vector product or a rank-1 update), i.e. we do not use padding. This strategy is implemented in M4RI and FFLAS/FFPACK.

We find that the number of additions is well-approximated by $2.141N^{\log_2 7}$ and the number of multiplications by $1.5N^{\log_2 7}$. We conclude that the constants hidden in the Big Oh are small and that it is not completely unreasonable to assume that they are equal to one. However, there is a noticeable thresholding effect.

Triangular matrix equations $UX = A$. We consider the problem of solving the matrix equation $UX = A$, when U is $n \times n$ and upper-triangular, while A is $n \times m$ (this can be seen as m independent triangular systems). The naive algorithm requires $2mn^2$ arithmetic operations (half multiplication, half additions).

The problem can be reduced to matrix multiplication. If n is odd, deal with the first coordinate naively (this requires $2mn$ operations). Then the process comes down to the solution of two triangular matrix equations of the same shape with size $n/2$ and a $(n \times n) \times (n \times m)$ matrix multiplication. Recursion stops when n goes below a given threshold.

We find $mn^{\log_2 7 - 1}$ multiplications and $1.33mn^{\log_2 7 - 1}$ additions.

LU factorization. Solving dense linear systems over finite fields (or computing a kernel basis) is dominated by LU factorization. There is a naive cubic algorithm, and a a block-recursive variant that reduces it to matrix multiplication and triangular system solving. This algorithm has the same asymptotic complexity as Strassen matrix multiplication.

We find that it requires $n^{\log_2 7}$ additions and $\frac{3}{4}n^{\log_2 7}$ multiplications. Again, there is a noticeable thresholding effect, as shown in Fig. 5.

4.2.2 Solving polynomial systems over "large" finite fields

We discuss in particular the case of $\mathbb{F}_{256}$.

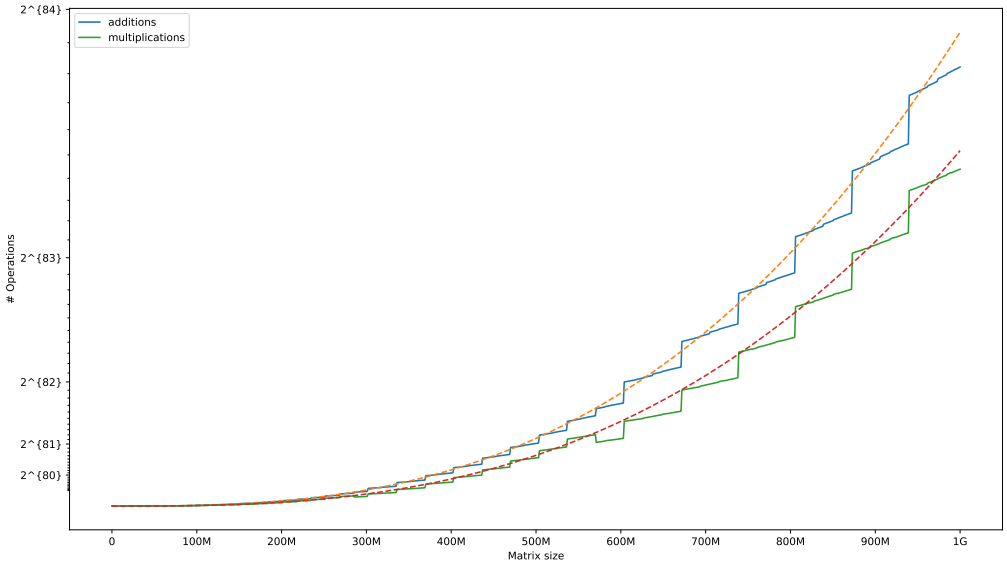


Figure 5: Cost of Solving Dense Linear Systems (computation of an LU factorization).

The XL algorithm. The XL algorithm was proposed by Courtois, Klimov, Patarin and Shamir [CKP+00] in 2000. In fact, it turned out to be a reinvention of a technique due to Lazard in 1983 [Laz83], and is more-or-less equivalent to modern Gröbner basis algorithms. What we say below about algorithmic and practical aspects is mostly based on the existing implementation of [CCN+12], that has been demonstrated to work and currently holds several computational records. It is capable of running a parallel computation using a cluster of machines. It was notably used in Beullens's practical cryptanalysis of Rainbow [Beu22]. This particular implementation works only over $\mathbb{F}_{16}$ and $\mathbb{F}_{31}$.

The underlying idea of the XL algorithm is simple. Pick a degree d such that the degree-d Macaulay matrix has full rank. Cut the column corresponding to the constant monomial. Call the resulting vector b and the truncated matrix A. Solve the linear system $Ax + b = 0$. If the original polynomial system had a single solution $\hat{x}$, then this linear system also has a single solution where the coordinates of x describe the values of all possible monomials of degree at most d, evaluated over $\hat{x}$. Note that this linear system is sparse, and can be solved using the block Wiedemann algorithm. Also, because we are only interested in the value of degree-1 monomials, it is sufficient to recover only a very small fraction of the solution vector. This allows the implementation of [CCN+12] to use an unpublished trick that bypasses the BW3 step almost completely.

The matrix A has $\binom{n+d}{d} - 1$ columns and $m\binom{n+d-2}{d-2}$ rows (it has much more rows than columns, and the rows have linear dependencies). The aforementioned implementation uses the following heuristic: a random subset of the rows is extracted to obtain a nearly square matrix, which is full-rank with high probability. Solving the input polynomial system is thus reduced to solving a sparse linear system of dimension $N := \binom{n+d}{d}$ with $\binom{n+2}{2}$ non-zero coefficients per row, using the block Wiedemann algorithm. Denoting by $\tilde{n}$ and $\tilde{m}$ the two blocking factors of the block Wiedemann algorithm, it follows from the discussion above that the total number of arithmetic

operations in the BW1 step is approximately:

$$2\left(1+\frac{\tilde{n}}{\tilde{m}}\right)\underbrace{\binom{n+d}{d}}_{N}^{2}\left(\binom{n+2}{2}+\tilde{m}\right)$$

To estimate the total number of operations, we ignore the costs of the BW2 and BW3 steps (the latter is almost zero), and assume that exactly N matrix-vector products take place. In other terms, we assume that $\tilde{n}/\tilde{m} \approx 0$ and that $\tilde{m} \ll n^2$. This gives a lower-bound on the number of operations.

Gröbner bases and the F_5 algorithm. Some of the best methods to solve systems of polynomial equations use Gröbner bases. A Gröbner basis of some polynomial ideal I is a set of generators of I that enjoy additional desirable properties. It is beyond the scope of this paper to discuss Gröbner bases; we refer the interested reader to a standard textbook such as [CLO91].

We will just point out that if the polynomial system $f_1 = \cdots = f_m = 0$ has a single solution $\hat{x}$ in the algebraic closure of the field, then this solution can easily be obtained by computing any Gröbner basis of the ideal spanned by the f_i's is$x_1 - \hat{x}_1, \ldots, x_n - \hat{x}_n$.

The classic method to compute such bases is known as Buchberger's algorithm [Buc65]. The state of the art, at this point, seems to be the F4 and F5 algorithms by Faugère [Fau99; Fau02]. F4 is essentially a reformulation of the Buchberger algorithm that does batch processing using efficient sparse linear algebra instead of polynomial manipulations. F5 strives to eliminate some useless computations. Faugère's algorithms have been successful in breaking some cryptosystems, most notably an instance of HFE with $n = 80$ variables, which turned to be spectacularly weak against Gröbner basis computations [FJ03].

Bardet described a simplified variant of the full-blown F_5 algorithm in [Bar04; BFS15] (this is called "matrix-F_5"). This algorithm computes an echelon form of the degree-d Macaulay matrix, using the F_5-criterion to efficiently discard rows that belong to the linear span of the rows above them. If the degree of regularity of the polynomial sequence is d, then it is commonly admitted that the F_5 algorithm terminates in less than

$$\mathcal{O}\left(\binom{n+d}{d}^{w}\right) \quad \text{operations over } \mathbb{F}_q. \tag{16}$$

The point is that a row-echelon form of a Macaulay matrix of sufficiently high degree reveals a Gröbner basis of the input polynomials. If they form a semi-regular sequence, it is sufficient to choose d equal to the degree of regularity. In fact, the matrix-F_5 algorithm processes a matrix of the same size as the XL algorithm.

While the original Macaulay matrices are quite sparse, this is no longer the case once they have been put into row echelon form. Special-purpose echelonization code and datastructures have been designed to perform this task [BEF+16]. In any case, the F_5 algorithm, or its simplified variants, always involve *dense* linear algebra. This has two consequences.

First, it is not reasonnable to assume that RREF computation can be done in quadratic time (at least, there is no known way to do so). Therefore, using $\omega = 2$ as the linear algebra constant is wrong with the F_5 algorithm. Second, the result of the computation (the RREF) may be fully dense, at least on non-pivotal columns, and therefore has large space requirements.

It follows that version 1.1.1 of the MQEstimator (using $\omega = 2$ by default) underestimated the cost of running the F_5 and the hybrid-F_5 algorithms by a huge margin. Version 1.2.0 of the

MQEstimator (using $\omega = 2.81$ by default) gives an accurate estimation of the behavior of the matrix-F_5 algorithm, but ignores the XL-Wiedemann algorithm.

However, forcibly setting $\omega = 2$ in MQEstimator this time leads to a (slight) underestimation of the running time: it is taken as N^2 where N is the size of the matrix, whereas it is in fact $\mathcal{O}\left(n^2N^2\right)$ using the block Wiedemann algorithm.

To the best of our knowledge, the only (serious) implementation of the full F_5 algorithm is `FGb` [Fau10]. It is closed-source and we do not have access to it. It further seems that it is not maintained and no longer available online.

There are competitive implementations of F_4, notably in the `msolve` (open-source) library [BES21] and in the `MAGMA` (closed-source) computer algebra software [BCP97]. Note that F_4 does not enjoy the same provable complexity guarantees as F_5. `MAGMA` is capable of solving polynomial systems over $\mathbb{F}_{256}$ (and most other finite fields). On the other hand, `msolve` can only compute a Gröbner basis over prime fields.

All these algorithms have exponential space complexity and existing implementation run into memory limitations even for a moderate number of variables. Implementing them is non-trivial, because they require either sophisticated data-structure for large-degree multivariate polynomials and/or sparse linear algebra over large matrices. Existing implementations are usually available inside full-blown computer algebra systems, which are large and complex software projects.

The hybrid method. The "hybrid method" [BFP09; BFP12] is usually the best technique for solving polynomial systems over finite fields. Its principle is simple:

1. Choose $0 \leq k \leq n$.
2. "Guess" the value of k variables.
3. Solve the remaining system of m equations $n - k$ variables using the F_5 of the XL-Wiedemann algorithm.
4. If no solution has been found, return to step 2.

The point is that the sub-systems that are actually solved in step 3 are more overdetermined than the input system, and therefore have a much lower degree of regularity. The resulting Macaulay matrices are thus much smaller.

There is an optimal number k of variables to guess. The asymptotic complexity of this procedure is determined in [BFP12] when $n \to +\infty$ with q and the ratio m/n fixed, under the assumption that the input system is sufficiently generic. Concretely, the optimal number of variables to guess depends on n, m, q and on the secondary algorithm used to solve the resulting polynomial systems.

The "polynomial XL" algorithm of [FK24] This algorithm can be seen as a (slight) generalization of Crossbred. In the end, it uses the hybrid method combined with the XL algorithm, but tries to perform a big precomputation to accelerate the subsequent resolution of polynomial systems.

It partitions the n input variables $x = (x_1, \ldots, x_n)$ in two categories. Say that they are relabeled as $x = (y_1, \ldots, y_k, z_1, \ldots, z_{n-k})$. The input polynomials are seen over the polynomial ring $\mathbb{F}_q[y][z]$, i.e. as polynomials in the z's whose coefficients are polynomials in the y's. The y's are the variable that will be "guessed", leading to a polynomial system in the z's.

The preprocessing step consists in finding at least α (linearly independent) degree-d polynomials in the ideal spanned by the f_i such that the total number of distinct monomials in the z's that appear in these new polynomials is at most α.

Once this is done, the y's are fixed to a random value, and the resulting system of α polynomial equations in the z's can be solved by linearization, by considering each of the possible α monomials as an independent variable.

The authors of Polynomial-XL (PXL) described a specific echelonization procedure to produce these α polynomials. Hence, this uses dense linear algebra on Macaulay matrices.

There is an effective way to predict the value of α, as well as the total number of field coefficient coefficients of the matrix after the end of the preprocessing. More precisely, assume that k is chosen and let d denote the degree of regularity, i.e. the smallest integer d such that $R_d(n-k,m) \geq \binom{n-k+d-1}{d}$, using the notations of section 4.2.1. Then $\alpha = \binom{n+d}{d} - \sum_{i=0}^{d} R_i(n-k,m)$.

In [FK24], the authors estimate the number of operation of their algorithm using either $\omega = 2.37$ or $\omega = 2.81$. We believe that only the second choice is reasonable. In this case, the gains claimed in [FK24] over the hybrid-XL-Wiedemann algorithm are modest (a factor of two for the largest examples).

What PXL and Crossbred have in common is that they have a preprocessing step (based on linear algebra in Macaulay matrices) that finds "special" polynomials in the ideal spanned by the input equation. These special polynomials are restricted to only have certain monomials.

In Crossbred, the only allowed monomials are those where the z's occur with degree at most d. Thus, once the y's are fixed, we are left with a degree-d polynomial system in $n-k$ variables. Existing practical implementations use $d = 1$, so the resulting linear system is small and easy to solve.

In PXL, the choice of allowed monomials is a bit more flexible, as there is no fixed upper-bound on their degree. The only condition is that their number must not be too large (once the y's are erased). In [FK24], the authors of PXL suggest a specific algorithm to select them, but in fact they could be chosen somewhat arbitrarily. In most cases, there is a value of d that yields almost the same number of polynomials with the Crossbred algorithm. The complexity of both algorithms therefore cannot be very different.

Parameters, estimations and discussion. Table 11 shows our choice of parameters for $q = 256$, along with our estimations of the complexity of running the (hybrid-)XL-Wiedemann (WXL) algorithm and the "Polynomial XL" (PXL) algorithm.

Even though an implementation of WXL is available, it is both quite difficult and a bit meaningless to predict its actual running time on a specific platform (e.g. "100 billion years on a single core of an Intel Xeon Gold 6230"), if only because the computation is meant to be infeasible. The actual obstacle to this more concrete estimate is the block-Wiedemann algorithm.

In the context of the Number Field Sieve, the block-Wiedemann algorithm has been executed in practice on a matrix of size 36M with 250 element per rows over a large finite field (to compute a discrete log), and on a matrix of size 400M with 250 elements per row over $\mathbb{F}_2$ (to factor RSA-250). Details of these computations are reported in [BGG+20]. In the context of the XL-Wiedemann, it was executed on a matrix of size 45M over $\mathbb{F}_{31}$. Details of the computation have been inferred by us, with some information available on the MQchallenge website. The BW1 step requires days of sequential processing (100, 18 and 19 days for the three described computations, respectively). By itself, this is a serious practical hurdle, and it

Level		I	III	V
n		48	72	96
WXL	k	2	4	7
	D	22	28	33
	$\log_2 N$	58.5	80	99
	$\log_2 \lvert A\rvert$	68.6	91.3	111.1
	cost	150	210	273
PXL	k	5	7	9
	D	17	24	30
	$\log_2 \alpha$	36	54.6	73
	$\log_2 \lvert A\rvert$	86.5	128.9	169
	cost	148	216	283

Table 11: Parameter choice with $q = 256$. For WXL, N denotes the size of the (sparse) matrix and $|A|$ denotes its number of non-zero coefficients. For PXL, α denotes the size of the (dense) matrix with polynomial entries resulting from the preprocessing and $|A|$ denotes its number of field coefficients. In both cases, $|A|$ is thus a reasonable estimate of the size of the matrix.

is not completely obvious that the algorithm "practically" scales to larger sizes. In [BGG+20], the authors conclude:

> [...] with adequate parameter choices, large sparse linear systems occurring in NFS computations can be handled, and at this point we are not facing a technology barrier.

However, the matrix sizes considered above are many orders of magnitude larger than those that have been dealt with in practice.

From a practical point of view, it is difficult to predict the actual running time (in hours) of the block-Wiedemann algorithm, even when the computation is practical. The process is well-known to be memory-bound or communication-bound, so the number of operations is not necessarily well-correlated to the actual running time. Choosing the blocking factors is not completely obvious either. If it is possible to measure the actual running time of one iteration, then the actual running time of the algorithm can be fairly well evaluated. However, predicting the time taken by the matrix-vector product is difficult: it depends on the hardware, on the shape of the matrix, on the clustering of entries inside it, etc.

4.2.3 Special case of Boolean systems

Estimating the difficulty of solving Boolean polynomial system is challenging because of the tension between "galactic" algorithms with the best asymptotic complexity and practically efficient ones.

Exhaustive search is the baseline method to solve systems of Boolean quadratic polynomial equations, with a running time $\tilde{\mathcal{O}}(2^n)$ and negligible space complexity. An FPGA implementation of exhaustive search [BCC+13] was used to break LUOV in practice [DDV+21].

Yang and Chen [YC04] discussed the asymptotic complexity of the hybrid method applied to Boolean system (along with the optimal number of variables to guess). The BooleanSolve algorithm of Bardet, Faugère, Salvy and Spaenlehauer [BFS+13] is the best embodiment of the hybrid method at this point, with running time $\tilde{\mathcal{O}}\left(2^{0.792n}\right)$ on average, under algebraic assumptions. It guesses some variables, then checks if a polynomial combination of the remaining polynomials is equal to 1. If it is the case, then the guessed values are incorrect (by Hilbert's Nullstellensatz). Checking this is accomplished by deciding whether large sparse linear systems have a solution (using the block-Wiedemann algorithm). The inventors of BooleanSolve claim that it is slower than exhaustive search when $n \leq 200$. However, this threshold should be treated with caution in the absence of an implementation.

The Crossbred algorithm of Joux and Vitse [JV17] also belongs to the "guess variables then solve a linear system" family of algorithms. Its asymptotic complexity is not precisely known, but its practical efficiency is spectacular: it has been used to solve record-size random systems with $n = 83$ variables and $m = 186$ equations, and with $m = 76$ equations in $n = 114$ variables. These are the current record. It is the first algorithm that has beaten brute-force in practice on random non-overdetermined systems. The original implementation by Joux and Vitse is not public. However, there are two public implementations: one that uses GPUs by Niederhagen, Ning and Yang [NNY18], and another more competitive one by Bouillaguet and Sauvage (`https://gitlab.lip6.fr/almasty/hpXbred` — this one holds the current records).

A completely different family of algorithms emerged in 2017 when Lokshtanov, Paturi, Tamaki, Williams and Yu [LPT+17] presented a randomized algorithm of complexity $\tilde{\mathcal{O}}\left(2^{0.8765n}\right)$ based on the "polynomial method". In strong contrast with almost all the previous ones, it does not require any assumption on the input polynomials, which is a theoretical breakthrough. The algorithm works by assembling a high-degree polynomial that evaluates to 1 on partial solutions, then approximates it by lower-degree polynomials. The technique was later improved by Björklund, Kaski and Williams [BKW19], reaching $\tilde{\mathcal{O}}\left(2^{0.804n}\right)$, then again by Dinur [Din21c], reaching $\tilde{\mathcal{O}}\left(2^{0.6943n}\right)$ — this is "Dinur's first algorithm".

Noting that the self-reduction that results in this low asymptotic complexity only kicks in for very large values of n, Dinur proposed a simpler, lightweight version of his algorithm for the crypto community with complexity $\mathcal{O}\left(n^2 2^{0.815n}\right)$ using $n^2 2^{0.63n}$ bits of memory [Din21a]. This one is known as "Dinur's second algorithm".

The main problem in choosing parameters is to estimate the number of operations of Dinur's algorithms (the first one in particular). It would be possible to "play safe" by choosing $n = \lambda/0.6943$, where λ is the desired security level, assuming that the hidden polynomial factors in the "big Oh tilde" are equal to one. This suggests choosing $n = 208$ for security level I. But in fact, the concrete number of operations required to run the algorithm is much higher than just $2^{0.6943n}$.

The crossbred algorithm. Because of its practical success, it seems fair to asses the efficiency of the Crossbred algorithm. We note that it is the first algorithm that has been capable of "beating brute force" in practice.

Just like Polynomial XL, the Crossbred algorithm partitions the n input variables $x = (x_1, \ldots, x_n)$ in two categories. Say that they are relabeled as $x = (y_1, \ldots, y_k, z_1, \ldots, z_{n-k})$. Its preprocessing step returns polynomials in which the z variables only occur with degree d. When the y variables are fixed to some value, we are left with a much smaller polynomial system that can be solved by linearization. In practical implementations, $d = 1$.

Finding these polynomials can be seen as finding vectors in the (left-)kernel of a Macaulay matrix in which some columns have been removed. It follows that there are essentially two

cases: if $d = 1$, then the number of polynomials that must be found is small, and they can be found efficiently by the block-Wiedemann algorithm. The Macaulay matrix remains in sparse representation and linear algebra is essentially quadratic. Otherwise, if $d \geq 2$, many kernel vectors must be found and dense Gaussian elimination is the only reasonable way to find them. In this case, the Macaulay matrix is in dense representation and linear algebra takes time $\mathcal{O}(N^\omega)$.

We consider that the CryptographicEstimators library has several shortcomings in its estimation of the complexity of Crossbred:

- It overestimates the space complexity by assuming a dense representation of the Macaulay matrix in the preprocessing step, even when it considers the possibility of using the Wiedemann algorithm (that allows the use of a sparse matrix). The `hpXbred` implementation uses a sparse matrix.
- It underestimate the running time of the exhaustive search phase by assuming that linear algebra runs in n^ω operations, even when the matrices are very small (when $d = 1$), when only the cubic algorithm makes sense.
- It overestimate the running time by ignoring the beneficial effect of external hybridation (it allows to increase the number of variables that are not exhaustively searched — see below).
- It overestimates the running time of the preprocessing step by assuming that the Wiedemann algorithm requires $3N$ matrix-vector products, when the block-Wiedemann algorithm with proper parameter choice ($m = 2n$, $n \geq 4$) can require $\leq 1.75N$.

Here is an example of a slight overestimation by the `CryptographicEstimators` library (`v2.0.0`, git commit `0d9bd3f925e`):

```
>>> from cryptographic_estimators import MQEstimator
>>> pb = MQEstimator.MQProblem(n=160, m=160, q=2)
>>> MQEstimator.Crossbred(pb).time_complexity()          # no external hybridation
151.16564211027884
>>> MQEstimator.Crossbred(pb, h=3).time_complexity()     # start by guessing 3 variables
150.43226492985866                                       # the result is better
```

We have some practical experience with the Crossbred algorithm, acquired by assembling the `hpXbred` high-performance implementation and using it to obtain all the current computational records of the MQChallenge website over $\mathbb{F}_2$. In the process, we have developped our own estimator (if only to choose parameters for actual computations).

An (unpublished) reduced-space hybrid method. It is well-known that, given a Boolean quadratic polynomial f, it is easy to find an invertible matrix S (a linear change of variables) such that $(Sx) = x_1x_2 + x_3x_4 + \cdots + x_{n-1}x_n + (\text{linear terms})$. If all the variables with odd index are "guessed", then $f(Sx)$ becomes linear. This allows to express one of the variables as a linear function of the others, and reduces the number of variables (and of polynomial equations) by one. It follows that a quadratic system with n equations in n variables can be solved by solving $2^{n/2}$ systems with $n - 1$ equations in $n/2 - 1$ variables.

This technique can be improved. It follows from [MPG13, Proposition 3] that, given *two* Boolean quadratic polynomials f and g, there is (often) a linear change of variables S such that

$f(Sx)$ and $g(Sx)$ are simultaneously *simplectic*. It follows that guessing half of the variables simultaneously turn $f(Sx)$ and $g(Sx)$ into linear functions. It follows that a quadratic system with n equations in n variables can be solved by solving $2^{n/2}$ systems with $n-2$ equations in $n/2-2$ variables.

This (unpublished) trick was used by the `hpXbred` implementation to obtain computational records in the "underdetermined" challenges category (where many variables can be "guessed" without reducing the success probability of the computation). The resulting subsystems are quite overdetermined, and the Crossbred algorithm performs well in this case. We call the resulting combination Crossbred$^+$. It always requires much less space than the "normal" Crossbred and is usually almost as fast.

Dinur's second algorithm. Because of the lack of any serious implementation, Dinur's algorithms pose the greatest challenge to a concrete estimation. This is probably not going to change because it seems likely that these algorithms cannot be competitive for any computation that can be carried out in practice. Dinur's second algorithm takes a parameter named n_1 in [Din21b]. Write:

$$N = \sum_{i=0}^{n_1+3} \binom{n-n_1}{i}$$

The algorithm has two dominating phases:

1. It needs to find all solutions of N quadratic systems of n_1+1 equations in n_1 variables (by brute force).
2. Then, a collection of n_1 polynomials of degree n_1+3 in $n-n_1$ variables must be interpolated and evaluated on all the 2^{n-n_1} possible inputs. Each such polynomial has N coefficients.

The space requirement of the algorithm is essentially $(n_1+1)N$ bits (to store these large polynomials). The value of n_1 is chosen to balance the costs of these two phases. The first phase is executed using the FES algorithm. To perform the second phase, Dinur described a memory-efficient version of the Moebius transform that evaluates a degree-d polynomial in n variables on all the 2^n possible inputs in time less than $n2^n$, using only twice the amount of memory needed to store the polynomial. We believe that the number of bit operations needed to complete the interpolation is quite underevaluated in [Din21b], for the following reason. It is stated in [Din21b] that:

> We estimate the complexity of a straight-line implementation of our algorithm by counting the number of bit operations (e.g., AND, OR, XOR) on pairs of bits. This ignores bookkeeping operations such as moving a bit from one position to another (which merely requires renaming of variables in straight-line programs).

In other terms, a statement such as:

$$A[2015494782137237151] \leftarrow A[8910305899308506505] \oplus A[14034715819129815024]$$

counts as a single bit operation. The time complexity of the attack is taken as the number of such statements. The problem is that this computational model is non-uniform: each statement representing a single bit operation contains three n-bit memory adresses that are "hard-coded" into the "code" of the procedure. The size of the procedure itself, measured in bits, is then larger than its number of statement by a factor of about $3n$. Generating the code of the procedure is

clearly more costly than executing it (in fact, the code may contain an "advice" of size linear in that of the input). At first glance, it is not really obvious how to compute the array indices "on the fly".

Bouillaguet [Bou24] has shown that the memory-efficient Möbius transform described by Dinur in [Din21b] can be implemented in the C language (with a single, fixed program for all possible input sizes) with a running time of $\mathcal{O}(d2^n)$ "elementary" operations on n-bit words (mostly additions). But this clearly requires more bit operations than the number claimed in [Din21b]. Experiments in [Bou24] suggest at least 20 CPU cycles per "bit operation". Therefore, we (optimistically) consider that the cost of the memory efficient Möbius transform is about $20nd2^n$ "gates" (this assumes that an elementary operation on n-bit words translates to exactly n gates).

In addition, we expect the space requirements to make the algorithm completely impractical.

Dinur's first algorithm. While having the best asymptotic complexity, Dinur's first algorithm suffer from a high concrete complexity for relevant instance sizes. In general, the algorithm returns the parity of the number of solutions ("does the polynomial system have an odd number of solutions?"). If we assume that the polynomial system has at most one solution, then this solves the decisional version of the problem ("does the system have a solution?"). The search-to-decision reduction "guesses" the first variable and checks if a solution still exists. It then proceeds with one less variable.

The algorithm is recursive, and each recursive calls need to choose a parameter (n_1 at the root, n_2 below). We searched for the best values exhaustively and implemented an estimator to determine its number of operations.

We applied the same penalty of a factor $20n$ to the Möbius transform as in Dinur's first algorithm.

We believe that, because the algorithm is complex and has never been implemented, any concrete estimation of its complexity should be taken with caution. Further "practical" improvements may be discovered if an implementation is ever attempted. However, the huge space complexity of the algorithm makes this unlikely.

Parameters, estimations and discussion. Using the `hpXbred` software, the running time of the Crossbred$^+$ algorithm on the level-I parameter set can be determined on currently existing hardware. It solves 2^{86} subsystems of 158 quadratic equations in 72 variables. Solving each subsystem requires 2375 CPU.h on a single machine (a PowerEdge C6420 blade equipped with two Intel Xeon Gold 6130 CPUs). This makes a total of 2×10^{25} CPU-years on this hardware platform. More precisely, the running times breaks down as follows:

1. BW1: 685 CPU.h
2. BW2: 25 CPU.h
3. BW3: 140 CPU.h
4. Enumeration: 1525 CPU.h

The matrix processed by the block-Wiedemann algorithm has dimension 8.86M and 6.5G non-zero entries. The BW1 and BW3 step total 2^{58} operations, that are executed at about 99.25Gop/s by the machine. The enumeration step has $2^{60.3}$ operations that are executed at

Level		I	III	V
n		160	240	320
Crossbred	h	3	4	3
	D	13	20	27
	k	31	44	56
	$\log_2 N$	61.7	95.9	130
	$\log_2 \lvert A \rvert$	75.3	111	145
	Cost	144	213	282
Crossbred$^+$	h	86	122	160
	D	5	8	11
	k	26	38	51
	$\log_2 N$	23.6	39.2	54.6
	$\log_2 \lvert A \rvert$	37.3	54	70.4
	Cost	147	216	285
Dinur 1st	n_1	39	58	80
	n_2	33, 0	52, 41, 33, 0	74, 61, 51, 43, 36, 31, 0
	Space	129	190	248
	Cost	160	223	283
Dinur 2nd	n_1	34	49	64
	Space	107	158	208
	Cost	148	214	280

Table 12: Parameter choice with $q = 2$.

260.6Gop/s by the machine. The difference is most likely explained by the fact that the block-Wiedemann algorithm suffers more from the cost of memory accesses. Note that the peak performance of a single core is about 2150Gop/s (2×512-bit AND per cycle at 2.1GHz).

5 Design choices

This section presents the design rationale of MQOM v2 in relation to the existing literature. Recent advancements in the field have led to signature schemes that are more efficient, more compact, and (sometimes) inherently simpler than their predecessors. We explain the rationale behind our design choices, which were made with an emphasis on simplicity in both design and implementation.

5.1 Threshold-Computation-in-the-Head

The design of MQOM v1 relied on the MPC-in-the-Head paradigm with additive sharings. Since then, two new frameworks have been introduced: the VOLE-in-the-Head (VOLEitH) framework [BBD+23] and the Threshold-Computation-in-the-Head (TCitH) framework [FR23b]. These new frameworks provide MQ-based signatures that are roughly half the size of MQOM v1 signatures, while also reducing computational costs. For these reasons, we decided to adopt one of these two frameworks in the development of MQOM v2.

TCitH vs. VOLEitH. While the line commitment scheme in TCitH-GGM only supports opening evaluations over a small domain Ω (with $|\Omega| = N$ being the size of the GGM tree), the VOLEitH framework supports evaluations over a domain of size N^τ by combining τ instances of the small-domain line commitment scheme. As a result, VOLEitH achieves a soundness error of $\frac{2}{N^\tau}$, compared to $\left(\frac{2}{N}\right)^\tau$ with the TCitH-based protocol repeated τ times. This implies that, for a given security level, τ can be slightly smaller in VOLEitH, often leading to reduced communication costs compared to TCitH-GGM.

On the other hand, the resulting large-domain line commitment scheme of VOLEitH requires a statistical consistency test, introducing an additional round of interaction between the prover and verifier in the underlying ZK PoK protocol. This slightly increases the overall communication and necessitates working over a large field extension (typically a λ-bit extended field). For signature schemes with a small secret witness –which is the case of the MQ solution x in MQOM–, this slight increase mitigates the TCitH overhead. As a consequence, the TCitH framework remains competitive in terms of signature size while enjoying a structurally simpler design.

In Table 13, we present the signature sizes for a variant of MQOM v2 based on the VOLEitH framework. We apply the same optimizations to this variant as in the TCitH-based version, including correlated trees and grinding (see the following subsections). Our results show that the short signature variants yield comparable sizes across both frameworks, while the fast signature variants are slightly smaller with the VOLEitH framework. Given this close proximity in signature size and the greater simplicity of the TCitH framework (no consistency check, smaller field extension), we ultimately chose to adopt the TCitH framework.

The sigma variant. In the considered PIOP protocol (see Section 2.1.2), the first verifier challenge is designed to batch multiple polynomials in order to reduce their communication cost. Instead of sending the m coordinates of the vector polynomial P_z (masked with P_u), the prover sends $\eta < m$ random linear combinations of them, that is the vector polynomial $\Gamma \cdot P_z$ (still masked with P_u, which is the vector polynomial P_α). We observed that the size saving due to this batching interaction is rather small for MQOM, in particular for the $\mathbb{F}_2$ instances. On the other hand, skipping the batching interaction results in a protocol with lower round

Framework	MQOM2 – TCitH		VOLEitH	
Statistical Batching	Without (3r)	With (5r)	Without (3r)	With (5r)
MQOM2-L1-gf2-short	2 868	2 820	2 966 (+3%)	2 822 (+0%)
MQOM2-L1-gf256-short	3 540	3 156	3 450 (-3%)	3 130 (-1%)
MQOM2-L1-gf2-fast	3 212	3 144	3 294 (+3%)	3 086 (-2%)
MQOM2-L1-gf256-fast	4 164	3 620	3 954 (-5%)	3 506 (-3%)
MQOM2-L3-gf2-short	6 388	6 280	6 788 (+6%)	6 428 (+2%)
MQOM2-L3-gf256-short	7 900	7 036	7 910 (+0%)	7 142 (+2%)
MQOM2-L3-gf2-fast	7 576	7 414	7 484 (-1%)	6 980 (-6%)
MQOM2-L3-gf256-fast	9 844	8 548	9 002 (-9%)	7 946 (-7%)
MQOM2-L5-gf2-short	11 764	11 564	12 170 (+3%)	11 498 (-1%)
MQOM2-L5-gf256-short	14 564	12 964	14 194 (-3%)	12 786 (-1%)
MQOM2-L5-gf2-fast	13 412	13 124	13 370 (-0%)	12 442 (-5%)
MQOM2-L5-gf256-fast	17 444	15 140	16 098 (-8%)	14 178 (-6%)

Table 13: Comparison of MQOM v2 with its variant over VOLEitH. All the signature size are in bytes.

complexity, 3 instead of 5, and arguably simpler design.[1] We have chosen to propose both options –with and without batching interaction– in the development of MQOM v2. Namely, we propose a 3-round variant (the "sigma variant") and a 5-round variant of MQOM. The 5-round variant features smaller signature sizes while the 3-round variant is simpler, easier to implement and could be more amenable in some specific contexts due to its lower round complexity. To the best of our knowledge, the sigma variant of MQOM v2 is the first MPCitH-based scheme built upon a sigma protocol (which does not rely on a protocol with helper [Beu20]).

Witness encoding as constant term versus as leading term. In the TCitH and VOLEitH frameworks, we encode the secret witness x in a polynomial P_x. There are two main options for encoding: either we construct P_x so that $P_x(0) = x$, encoding the witness as the constant term, or we define P_x so that $P_x(\infty) = x$, encoding the witness as the leading term.

The TCitH framework [FR23b] originally suggested encoding the witness as the constant term, which is the most traditional approach in the literature when using Shamir's secret sharing. In contrast, the VOLEitH framework suggests encoding the witness as the leading term of the (degree-1) polynomial. However, both frameworks do not mandate a specific encoding; we can choose to encode the witness as the leading term in TCitH and as the constant term in VOLEitH.

Each option has its advantages and disadvantages. Encoding the witness as the constant term requires special handling to avoid evaluation at the zero point, and we must use "infinity" point when $N = |\mathbb{F}|$. It also necessitates dealing with the inversion of some publicly known field elements. On the other hand, encoding the witness as the leading term results in a simpler implementation since no inversion is required and we avoid the special "infinity" point when $N = |\mathbb{F}|$. This makes it possible to use the canonical injection of $\{0, \ldots, N-1\}$ into field elements for Ω. However, this approach results in a slightly higher number of field multiplications,

[1] With the VOLEitH framework, skipping the batching interaction also lowers the round complexity from 7 to 5.

as we need to account for the homogeneous form of the MQ constraints. For the sake of implementation simplicity, we have chosen to encode the witness as the *leading term* of P_x in MQOM v2.

5.2 GGM trees

Recent improvements have made the arithmetic part of MPCitH-based signature schemes more efficient, shifting the computational and communication bottleneck to the symmetric part, particularly GGM trees. Consequently, several recent works have focused on optimizing the symmetric part, primarily by improving all-but-one vector commitments based on GGM trees.

Half-Tree technique [GYW+23; CLY+24; BCD24]. The half-tree technique has been proposed in [GYW+23] and has been first introduced in to the MPCitH context in [CLY+24; BCD24]. This technique aims to optimize the tree derivation, *i.e.* how two children nodes are generated from the parent node. Instead of using a double-length pseudorandom generator, it consists of deriving the first child y from the parent x using a symmetric primitive and building the second child z as $z := x \oplus y$, where $\oplus$ is the XOR operation. This derivation maintains the core security property of tree derivation: revealing one child node should not disclose any information about its sibling. Specifically, if revealing y does not leak information about x, then it also does not reveal anything about $z = y \oplus x$, as x masks it. Likewise, revealing z does not disclose any information about $y = x \oplus z$.

In MQOM v2, we use the half-tree optimization to halve the cost of the tree derivation and because it unlocks a second optimization described below.

Correlated Trees [HJ24; KLS24]. Besides the computational advantage of the half-tree technique, the latter has an interesting property: a tree derivation when using the half-tree preserves the XOR. It implies that the XOR of all the nodes at a same depth is the same across the entire tree. This means that the committer can control the XOR-sum δ of all the leaf seeds (by originally introducing this difference between the two child nodes of the root). Then, replacing the pseudorandom tape $\mathsf{PRG}(\mathsf{seed})$ by $\mathsf{seed} \,\|\, \mathsf{PRG}(\mathsf{seed})$, the XOR-sum δ is further enforced to the λ first bits of the random tapes. This makes it possible to save λ bits of communication in the correction value Δ_x by fixing δ to the λ first bits of x, thus leading to shorter signatures.

In MQOM v2, we use this optimization, namely we use correlated trees to save $\tau \cdot \lambda$ bits in the signature.

One-tree optimization [BBM+24]. Some combinatorial optimizations of the GGM trees has been proposed in [BBM+24]. The MPCitH/TCitH/VOLEitH-based schemes always use τ GGM trees. For each of them all the leaves except one are opened. Instead of considering τ independent GGM trees of N leaves in parallel, the authors of [BBM+24] suggest using a unique large GGM tree of $\tau \cdot N$ leaves. The i^{th} leaf of the e^{th} tree becomes the $(e \cdot N + i)^{\text{th}}$ leaf of the large unique tree. As explained in [BBM+24], "opening all-but-τ leaves of the big tree is more efficient than opening all-but-one leaves in each of the τ small trees because with high probability some of the active paths in the tree will merge relatively close to the leaves, which reduces the number of internal nodes that need to be revealed." Then, the authors propose to improve the previous point using some rejection sampling and grinding. When the last Fiat-Shamir challenge is such that the number of releaved nodes in the revealed sibling paths exceeds a threshold, the signer rejects the challenge and recompute the hash with an incremented counter. This process is

repeated until the number of revealed nodes is below the fixed threshold. One can show that the approach leads to secure scheme even if the challenge space is reduced, because the security loss is compensated by the computational cost of searching a valid challenge.

Let us stress that this optimization is not compatible with the correlated-tree optimization. In MQOM v2, we did not consider this optimization because it complicates the design and implementation. First, using this large tree of $\tau \cdot N$ prevents from using a complete binary tree (since τ is usually not a power of 2), and so one needs to handle leaves of different depths. Then, while conceptually easy to understand, this optimization requires dealing with path merging which is tricky in terms of implementation. Finally, while path merging saves communication, it introduces variability in the signature size, which we prefer to avoid.

Relaxed vector commitment [KLS24]. A recent work [KLS24] proposes a new idea to slightly improve the efficiency of GGM-tree all-but-one vector commitments. It consists in relaxing the vector commitment scheme by committing each leaf of the GGM trees using λ-bit digests instead of 2λ-bit digests. While this relaxation breaks the standard notion of binding, the authors show that using such a relaxed commitment scheme within a signature scheme still leads to the desired security. The high-level principle is the following: instead of properly binding the tree leaves, this relaxed commitment scheme *binds the height-1 nodes* (the parent nodes of the leaves) using the fact that the 2 (λ-bit) commitment digests of the two children form a 2λ-bit commitment digest for the parent node, which prevents the prover to get collisions over those nodes. Moreover, the authors show that the prover can have at most $2\lambda/\log_2 \lambda$ *preimages of the leaf commitment*. By counting the number of possible openings, the authors show that the relaxed vector commitment can be opened to at most $u := 2N(\lambda/\log 2\lambda)^2$ different witnesses. This relaxed opening degrades the soundness of the proof system by an offset of $\log_2 u$ bits. This security loss can be compensated by increasing the scheme parameters while still benefitting from a decreased signature size.

We did not include this optimization in MQOM v2 but we will consider it for a future update after careful analysis of its security and adaptation to the TCitH context. We could expect a saving of around 200 bytes in the signature size (for the first security level).

5.3 Grinding

Together with the one-tree optimization, [BBM+24] suggests using an explicit proof-of-work to the Fiat-Shamir hash computation of the query challenge, whihc is known as *grinding* [Sta21]. Together with the query challenge, the signer samples a w-bit value which should be zero, for w a parameter of the scheme. If this value is not zero, the signer rejects the query challenge and recompute the hash with an incremented counter, until a zero value is found. This strategy increases the cost of hashing the last challenge by a factor 2^w which translates to decreasing the soundness error by a factor 2^{-w}. As a consequence, one can lower the GGM tree parameters to achieve a soundness of $\lambda - w$ bits instead of λ.

We use the grinding optimization in MQOM v2. We chose the grinding parameter w in order to decrease the number of repetitions (τ).

5.4 Symmetric primitives

For most of the symmetric primitives involved in the signature scheme, there are two possible options to instantiate them: either we use an extendable output hash function (XOF), or we use

a block cipher. While the first version of MQOM solely relied on XOF, we changed for mainly using a block cipher in MQOM v2 for performance reasons. Specifically, we aim to leverage the AES hardware instructions available on modern CPUs. While the easiest choice would then be to use AES-λ as block cipher (e.g. as counter mode for the PRG), this choice implies a 128-bit distinguisher whatever λ because of the fixed 128-bit block-size of AES. As a result, the EUF-CMA advantage can only be upper bounded by 2^{-128} whatever the target security λ.

To avoid this issue, we use a block cipher with higher block size for Categories III and V, namely Rijndael-256-256 (with truncation for $\lambda = 192$). The latter cipher also benefits from fast implementation using AES hardware instructions. Moreover, using a cipher with a λ-bit key-size and λ-bit block-size, we can use a (partial) fixed-key mode in the seed derivation with a Davies-Meyer construction. This further improves the performances by saving some costly key schedules.

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 2 and 14 magacycles (0.9 – 5.6 ms AMD Ryzen Threadripper PRO 7995WX 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.
- **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 (arithmetic computation, 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.
- **Relatively small signatures:** The last generation of MPCitH-based signature schemes in the literature (at time of writing) has signature sizes ranging on 2.5–5 KB (for 128-bit of security). MQOM is on the lower side of this range, with 2.8–3.6 KB. Moreover, MPCitH-based signatures achieving lower sizes are based on arguably less conservative assumptions such as *e.g.* recent dedicated symmetric designs.

References

[AB24] M. Albrecht and G. Bard. *The M4RI Library – Version **version**.* The M4RI Team. 2024 (cited on page 37).

[AW21] J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication. In D. Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021 (cited on page 37).

[Bar04] M. Bardet. *Étude des systèmes algébriques surdéterminés. Applications aux codes correcteurs et à la cryptographie.* PhD thesis, Pierre and Marie Curie University, Paris, France, 2004 (cited on pages 38, 39, 42).

[BBD+23] C. Baum, L. Braun, C. Delpech de Saint Guilhem, M. Klooß, E. Orsini, L. Roy, and P. Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head. In pages 581–615, 2023 (cited on pages 1–3, 8, 52).

[BBM+24] C. Baum, W. Beullens, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. One tree to rule them all: optimizing GGM trees and OWFs for post-quantum signatures. In *ASIACRYPT 2024, Part I*, pages 463–493, 2024 (cited on pages 54, 55).

[BCC+13] C. Bouillaguet, C. Cheng, T. Chou, R. Niederhagen, and B.-Y. Yang. Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs. In *Selected Areas in Cryptography*, pages 205–222. Springer, 2013. https://eprint.iacr.org/2013/436.pdf (cited on page 45).

[BCD24] D. Bui, K. Cong, and C. Delpech de Saint Guilhem. Improved all-but-one vector commitment with applications to post-quantum signatures. Cryptology ePrint Archive, Report 2024/097, 2024 (cited on page 54).

[BCP97] W. Bosma, J. J. Cannon, and C. Playoust. The Magma Algebra System I: The User Language. *J. Symb. Comput.*, (3/4):235–265, 1997 (cited on page 43).

[BEF+16] B. Boyer, C. Eder, J.-C. Faugère, S. Lachartre, and F. Martani. Gbla: gröbner basis linear algebra package. In *Proceedings of the 2016 ACM International Symposium on Symbolic and Algebraic Computation*, 135–142, Waterloo, ON, Canada. Association for Computing Machinery, 2016. ISBN: 9781450343800 (cited on page 42).

[BES21] J. Berthomieu, C. Eder, and M. Safey El Din. msolve: A Library for Solving Polynomial Systems. In *2021 International Symposium on Symbolic and Algebraic Computation*, pages 51–58, Saint Petersburg, Russia. ACM, 2021 (cited on page 43).

[Beu20] W. Beullens. Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part III*, pages 183–211, 2020 (cited on page 53).

[Beu22] W. Beullens. Breaking rainbow takes a weekend on a laptop. In Y. Dodis and T. Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*, pages 464–479. Springer, 2022 (cited on page 41).

[BFP09] L. Bettale, J. Faugère, and L. Perret. Hybrid approach for solving multivariate systems over finite fields. *J. Math. Cryptol.*, (3):177–197, 2009 (cited on page 43).

[BFP12] L. Bettale, J. Faugère, and L. Perret. Solving polynomial systems over finite fields: improved analysis of the hybrid approach. In J. van der Hoeven and M. van Hoeij, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC'12, Grenoble, France - July 22 - 25, 2012*, pages 67–74. ACM, 2012 (cited on page 43).

[BFR24] R. Benadjila, T. Feneuil, and M. Rivain. MQ on my mind: post-quantum signatures from the non-structured multivariate quadratic problem. In pages 468–485, 2024 (cited on pages 1, 36).

[BFS15] M. Bardet, J. Faugère, and B. Salvy. On the complexity of the F5 gröbner basis algorithm. *J. Symb. Comput.*:49–70, 2015 (cited on pages 38, 42).

[BFS+13] M. Bardet, J. Faugère, B. Salvy, and P. Spaenlehauer. On the complexity of solving quadratic boolean systems. *J. Complexity*, (1):53–75, 2013 (cited on page 46).

[BGG+20] F. Boudot, P. Gaudry, A. Guillevic, N. Heninger, E. Thomé, and P. Zimmermann. Comparing the difficulty of factorization and discrete logarithm: A 240-digit experiment. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part II*, pages 62–91, 2020 (cited on pages 39, 44, 45).

[BKW19] A. Björklund, P. Kaski, and R. Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. In C. Baier, I. Chatzigiannakis, P. Flocchini, and S. Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, 26:1–26:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019 (cited on page 46).

[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, *Progress in Cryptology - AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18-20, 2022, Proceedings*, pages 323–347. Springer Nature Switzerland, 2022 (cited on page 36).

[Bou24] C. Bouillaguet. Algorithm xxx: evaluating a boolean polynomial on all possible inputs. *ACM Trans. Math. Softw.*, 2024. Just Accepted (cited on page 49).

[Buc65] B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (An Algorithm for Finding the Basis Elements in the Residue Class Ring Modulo a Zero Dimensional Polynomial Ideal)*. PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965. English translation in J. of Symbolic Computation, Special Issue on Logic, Mathematics, and Computer Science: Interactions. Vol. 41, Number 3-4, Pages 475–511, 2006 (cited on page 42).

[CCN+12] C. Cheng, T. Chou, R. Niederhagen, and B. Yang. Solving quadratic equations with XL on parallel architectures. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, pages 356–373. Springer, 2012 (cited on pages 39, 41).

[CDI05] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *TCC 2005*, pages 342–362, 2005 (cited on page 8).

[CKP+00] N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In B. Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 392–407. Springer, 2000 (cited on page 41).

[CLO91] D. A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, (Undergraduate Texts in Mathematics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991. ISBN: 0387356509 (cited on page 42).

[CLY+24] H. Cui, H. Liu, D. Yan, K. Yang, Y. Yu, and K. Zhang. ReSolveD: shorter signatures from regular syndrome decoding and VOLE-in-the-head. In *PKC 2024, Part I*, pages 229–258, 2024 (cited on page 54).

[DDV+21] J. Ding, J. Deaton, Vishakha, and B. Yang. The nested subset differential attack - A practical direct attack against LUOV which forges a signature within 210 minutes. In A. Canteaut and F. Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, pages 329–347. Springer, 2021 (cited on page 45).

[DGP08] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Trans. on Mathematical Software (TOMS)*, (3):1–42, 2008 (cited on page 37).

[Din21a] I. Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In A. Canteaut and F. Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, pages 374–403. Springer, 2021 (cited on page 46).

[Din21b] I. Dinur. Cryptanalytic applications of the polynomial method for solving multivariate equation systems over GF(2). In A. Canteaut and F.-X. Standaert, editors, *EUROCRYPT 2021, Part I*, pages 374–403, 2021 (cited on pages 48, 49).

[Din21c] I. Dinur. Improved algorithms for solving polynomial systems over GF(2) by multiple parity-counting. In D. Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 2550–2564. SIAM, 2021 (cited on page 46).

[EVZ+24] A. Esser, J. A. Verbel, F. Zweydinger, and E. Bellini. Sok: cryptographicestimators - a software library for cryptographic hardness estimation. In J. Zhou, T. Q. S. Quek, D. Gao, and A. A. Cárdenas, editors, *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. ACM, 2024 (cited on page 36).

[Fau02] J.-C. Faugère. A New Efficient Algorithm for Computing Gröbner Bases Without Reduction to Zero (F5). In T. Mora, editor, *ISSAC '02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pages 75–83, Lille, France. ACM Press, 2002. ISBN: 1-58113-484-3. isbn: 1-58113-484-3 (cited on page 42).

[Fau10] J. Faugère. Fgb: A library for computing gröbner bases. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, pages 84–87. Springer, 2010 (cited on page 43).

[Fau99] J.-C. Faugère. A new efficient algorithm for computing grobner bases (f4). *Journal of Pure and Applied Algebra*, (1-3):61–68, 1999 (cited on page 42).

[FJ03] J. Faugère and A. Joux. Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using gröbner bases. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 44–60. Springer, 2003. ISBN: 3-540-40674-3 (cited on page 42).

[FK24] H. Furue and M. Kudo. Polynomial XL: A variant of the XL algorithm using macaulay matrices over polynomial rings. In M. O. Saarinen and D. Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Oxford, UK, June 12-14, 2024, Proceedings, Part II*, pages 109–143. Springer, 2024 (cited on pages 43, 44).

[FR23a] T. Feneuil and M. Rivain. MQOM: MQ on my Mind – Algorithm Specifications and Supporting Documentation. Version 1.0 – 31st May 2023, 2023. https://mqom.org/docs/mqom-v1.0.pdf (cited on pages 1, 36).

[FR23b] T. Feneuil and M. Rivain. Threshold computation in the head: improved framework for post-quantum signatures and zero-knowledge arguments. Cryptology ePrint Archive, Report 2023/1573, 2023 (cited on pages 1–3, 8, 52, 53).

[FR23c] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In *ASIACRYPT 2023, Part I*, pages 441–473, 2023 (cited on page 2).

[GKW+20] C. Guo, J. Katz, X. Wang, and Y. Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE Computer Society Press, 2020 (cited on page 12).

[GYW+23] X. Guo, K. Yang, X. Wang, W. Zhang, X. Xie, J. Zhang, and Z. Liu. Half-tree: halving the cost of tree expansion in COT and DPF. In C. Hazay and M. Stam, editors, *EUROCRYPT 2023, Part I*, pages 330–362, 2023 (cited on pages 9, 54).

[HJ24] J. Huth and A. Joux. MPC in the head using the subfield bilinear collision problem. In *CRYPTO 2024, Part I*, pages 39–70, 2024 (cited on page 54).

[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).

[JV17] A. Joux and V. Vitse. A Crossbred Algorithm for Solving Boolean Polynomial Systems. In *NuTMiC*, pages 3–21. Springer, 2017. https://eprint.iacr.org/2017/372.pdf (cited on page 46).

[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, 7).

[KLS24] S. Kim, B. Lee, and M. Son. Relaxed vector commitment for shorter signatures. Cryptology ePrint Archive, Report 2024/1004, 2024 (cited on pages 36, 54, 55).

[Laz83] D. Lazard. Gröbner-bases, gaussian elimination and resolution of systems of algebraic equations. In J. A. van Hulzen, editor, *EUROCAL*, pages 146–156. Springer, 1983. ISBN: 3-540-12868-9 (cited on pages 38, 41).

[LPT+17] D. Lokshtanov, R. Paturi, S. Tamaki, R. R. Williams, and H. Yu. Beating brute force for systems of polynomial equations over finite fields. In P. N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017. ISBN: 978-1-61197-478-2 (cited on page 46).

[MPG13] G. Macario-Rat, J. Plût, and H. Gilbert. New insight into the isomorphism of polynomial problem IP1S and its use in cryptography. In K. Sako and P. Sarkar, editors, *ASIACRYPT 2013, Part I*, pages 117–133, 2013 (cited on page 47).

[NIS22] N. I. of Standards and T. (NIST). Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process, 2022. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf (cited on page 1).

[NNY18] R. Niederhagen, K. Ning, and B. Yang. Implementing joux-vitse's crossbred algorithm for solving MQ systems over GF(2) on gpus. In T. Lange and R. Steinwandt, editors, *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, pages 121–141. Springer, 2018 (cited on page 46).

[Roy22] L. Roy. SoftSpokenOT: quieter OT extension from small-field silent VOLE in the minicrypt model. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022, Part I*, pages 657–687, 2022 (cited on page 8).

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Report 2021/582, 2021 (cited on pages 11, 55).

[SZ22] A. Szepieniec and Y. Zhang. Polynomial IOPs for linear algebra relations. In G. Hanaoka, J. Shikata, and Y. Watanabe, editors, *PKC 2022, Part I*, pages 523–552, 2022 (cited on page 2).

[Tha23] J. Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2023. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf (cited on page 2).

[YC04] B. Yang and J. Chen. Theoretical analysis of XL over small fields. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, pages 277–288. Springer, 2004 (cited on page 46).

[YSW+21] K. Yang, P. Sarkar, C. Weng, and X. Wang. QuickSilver: efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 2986–3001. ACM Press, 2021 (cited on page 3).