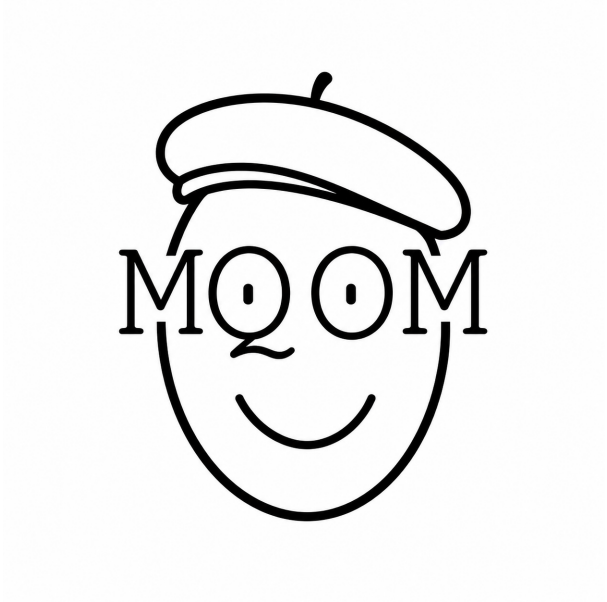
MQOM: MQ on my Mind

Algorithm Specifications and Supporting Documentation (Version 3.0)

Ryad Benadjila Charles Bouillaguet
Thibauld Feneuil Matthieu Rivain

August 31, 2026

CryptoExperts, Sorbonne Université



Contents

Changelog

2026-08-31: Version 2.1 → Version 3.0

We have added a variant based on the one-tree optimization while retaining the correlated-tree variant. Although this new variant enjoys a tighter security reduction to the multivariate quadratic (MQ) problem, it offers similar signature sizes but is generally less implementation-friendly.

To improve the overall performance, we made the following changes:

- The public MQ equations are now expanded from seeds using the block cipher in counter mode instead of the Davies-Meyer construction.
- Rather than deriving all batching challenges from a single global hash, we now derive the batching challenge of each parallel repetition solely from the commitment of that repetition. This modification is sound because the batching soundness error is already negligible for each individual repetition.
- We reordered the inputs of several hash computations, modified parts of the hash chain construction, and updated the transcript format to improve streamability and reduce memory usage.
- We replaced the hash-based grinding procedure with an AES-based one.

To further strengthen the security of the scheme, we made the following changes:

- The seed expansion of the correlated-tree variant has been modified to eliminate the need for the RGOW-MQOM assumption, following the approach proposed in [FR26].
- Several domain-separation issues have been fixed, including the salt-less root-seed expansion issue identified in [Del26].
- The MQ parameters over $\mathbb{F}_{16}$ have been increased to provide an additional security margin against potential refinements of attacks on the MQ problem over this field, while favoring parameter choices that are more implementation-friendly.

As a consequence, the *known answer tests* have been updated.

We also introduced a new variant over $\mathbb{F}_2$, labelled shorter, targeting applications where signature size is the primary concern. To simplify the parameter sets, we removed the instances over the field $\mathbb{F}_{256}$ and previous instances over $\mathbb{F}_2$ (i.e., all $\mathbb{F}_2$ instances except the new shorter variant), as well as all the three-round variants (i.e. the variants skipping the batching round, labelled `*-r3` in Version 2). We now provide three variants: fast (over $\mathbb{F}_{16}$), short (over $\mathbb{F}_{16}$), and shorter (over $\mathbb{F}_2$), each for both the correlated-tree optimization (labelled `*-ct`) and the one-tree optimization (labelled `*-ot`), for the three security categories (I, III, and V). This makes a total of 18 instances.

Finally, we expanded the security section to discuss security proofs and recent attack paths, included benchmark results for improved implementations targeting both `x86` platforms and embedded devices, and added a new section describing efficient implementation techniques for MQOM.

2025-09-22: Version 2.0 → Version 2.1

We have replaced the field-embedding batching with the packing strategy. While both approaches are syntactically equivalent from a security perspective, packing allows us to shift the arithmetic overhead from $\mathbb{F}$ to $\mathbb{K}$. We have also updated the definition of the evaluation domain Ω: instead of relying on field elements ordered lexicographically, we now use Gray code, which improves the efficiency of the folding step. Although these two changes do not affect the security analysis of the scheme, they modify the structure of the public key and the signature. Consequently, the *known answer tests* have been updated.

We have also introduced a new scheme variant whose security relies on the hardness of solving the multivariate quadratic problem over the field $\mathbb{F}_{16}$. In contrast, the previous variants relied on $\mathbb{F}_2$ and $\mathbb{F}_{256}$. This new variant offers both competitive signature sizes and running times: the $\mathbb{F}_2$ variant suffers from limited computational performance, while the $\mathbb{F}_{256}$ variant incurs relatively large signature sizes.

Finally, we have provided benchmark results for new highly-optimized implementations targeting both `x86` platforms and embedded devices.

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 v3, the third version of the MQOM signature scheme, a third round candidate to the NIST *call for additional digital signature schemes for the post-quantum cryptography standardization process* [NIS22].

Similarly to MQOM v2, the proof system in MQOM v3 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/MQOM v3 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. MQOM v3 then achieves signatures of 2.5–3.3 KB.

Unless otherwise specified, MQOM should refer to MQOM v3 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 discusses several implementation aspects. Section 5 provides a security analysis of the MQOM signature scheme. Finally, Section 6 discusses the main design choices of MQOM, while Section 7 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 challenges 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 guarantee that some polynomials 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].

Packing. Let $\mathbb{K}$ be a degree-μ extension field of $\mathbb{F}$. Let $\beta_1, \ldots, \beta_\mu$ be an $\mathbb{F}$-basis of $\mathbb{K}$ and let

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

be the associated $\mathbb{F}$-linear field-embedding isomorphism.

The MQOM PIOP performs computation in the field extension $\mathbb{K}$, and its efficiency depends on the number of degree-2 constraints over $\mathbb{K}$. The above constraint $F(x) = 0$ corresponds to m degree-2 constraints over $\mathbb{F}$. To reduce the number of constraints, we apply a packing technique similar to that in [BBM+24]. Specifically, we define:

$$\begin{array}{rclcrcl}
\hat{A}_1 & = & \phi(A_1, \ldots, A_\mu) \ , & \ldots & \hat{A}_{\hat{m}} & = & \phi(A_{m-\mu+1}, \ldots, A_m) \ , \\
\hat{b}_1 & = & \phi(b_1, \ldots, b_\mu) \ , & \ldots & \hat{b}_{\hat{m}} & = & \phi(b_{m-\mu+1}, \ldots, b_m) \ , \\
\hat{y}_1 & = & \phi(y_1, \ldots, y_\mu) \ , & \ldots & \hat{y}_{\hat{m}} & = & \phi(y_{m-\mu+1}, \ldots, y_m) \ ,
\end{array}$$

with $\hat{m} = m/\mu$ (assuming m is a multiple of μ). Instead of proving that $x \in \mathbb{F}^n$ satisfies $F(x) = (0, \ldots, 0)$, the MQOM PIOP proves that x satisfies $\hat{F}(x) = (0, \ldots, 0) \in \mathbb{K}^{\hat{m}}$, where

$$\hat{F} = (\hat{f}_1, \ldots, \hat{f}_{\hat{m}}) \quad \text{with} \quad \hat{f}_i : x \mapsto x^\intercal \hat{A}_i x + \hat{b}_i^\intercal x - \hat{y}_i \ . \tag{2}$$

By $\mathbb{F}$-linearity of ϕ, this new statement is strictly equivalent to the original one, but it involves $\hat{m} = m/\mu$ degree-2 constraints over $\mathbb{K}$ instead of m degree-2 constraints over $\mathbb{F}$. To avoid repeated conversions, MQOM key generation outputs the MQ instance directly in packed form $(\hat{A}_i, \hat{b}_i, \hat{y}_i)_{i \in [\hat{m}]}$, and the signing and verification algorithms operate exclusively on this packed representation.

Notions. Let $\Omega \subseteq \mathbb{K}$ 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 $\hat{F}$ to a degree-1 vector polynomial $P \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^n$, which results in a degree-2 vector polynomial $\hat{F}(P) \in \left(\mathbb{K}[X]^{\leqslant 2}\right)^{\hat{m}}$ such that

$$\hat{F}(P)(X) = (P(X)^\intercal \hat{A}_i P(X) + \hat{b}_i^\intercal P(X) \cdot X - \hat{y}_i \cdot X^2)_i \ .$$

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 $\hat{F}(P) \in \mathbb{K}[X]^{\leqslant 2}$, $\hat{F}(P)(\infty)$ 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 $\hat{F}(P)(\infty) = \hat{F}(x)$, which is the all-0 vector if and only if x is the MQ solution associated to F (and $\hat{F}$).

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 $P_u \in \left(\mathbb{K}[X]^{\leqslant 1}\right)^{\hat{m}}$ 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 computes $P_\alpha = P_u + \hat{F}(P_x)$ 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 $P_\alpha(r) = P_u(r) + \hat{F}(P_x(r))$.

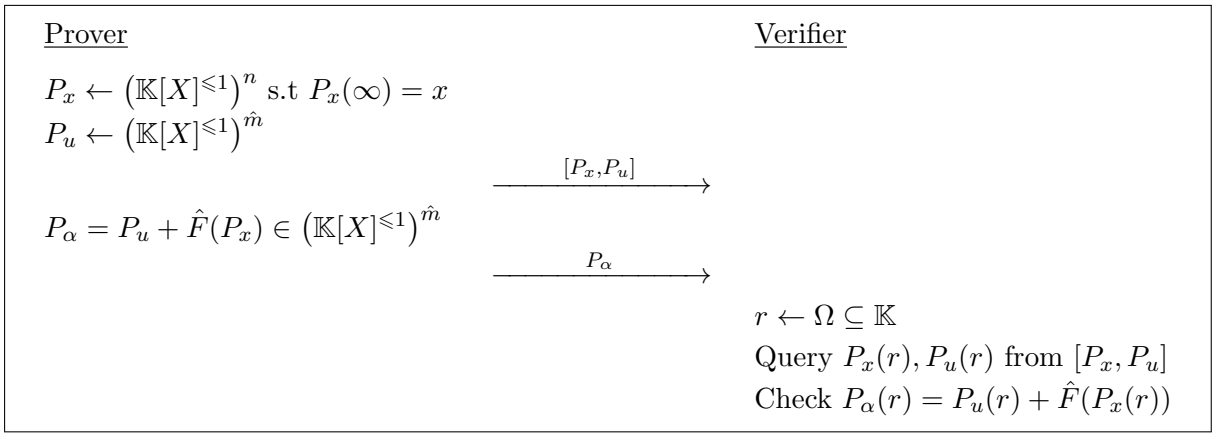


Figure 1: Polynomial IOP underlying MQOM (simple version).

The vector polynomial $P_z := \hat{F}(P_x)$ is of degree 1 if and only if $P_z(\infty) = \hat{F}(P_x(\infty)) = (0, \ldots, 0) \in \mathbb{K}^{\hat{m}}$, 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 $P_\alpha(r) = P_u(r) + \hat{F}(P_x(r))$ holds for 3 different points $r \in \Omega$, then because P_x, P_u, P_α are guaranteed to be of degree 1, we must have $P_\alpha = P_u + \hat{F}(P_x)$ and hence $\hat{F}(x) = 0$ (i.e. the prover indeed knows a solution x). Conversely, in the presence of a malicious prover (who does not know a correct solution x), we can only have $P_\alpha(r) = P_u(r) + \hat{F}(P_x(r))$ 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.

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 neither impacts the soundness nor the communication, choosing the leading term has some advantages in terms of computation (see Section 6 for further discussion).*

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

Batching with random combinations. To 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 $\hat{m}$ coordinates of $\hat{F}(P_x)$ into η random linear combinations for some parameters $\eta \in \mathbb{N}$. Let $\Gamma \in \mathbb{K}^{\eta \times \hat{m}}$ be a matrix randomly sampled by the verifier. The prover now defines P_α as:

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

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

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

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 batching strategy.

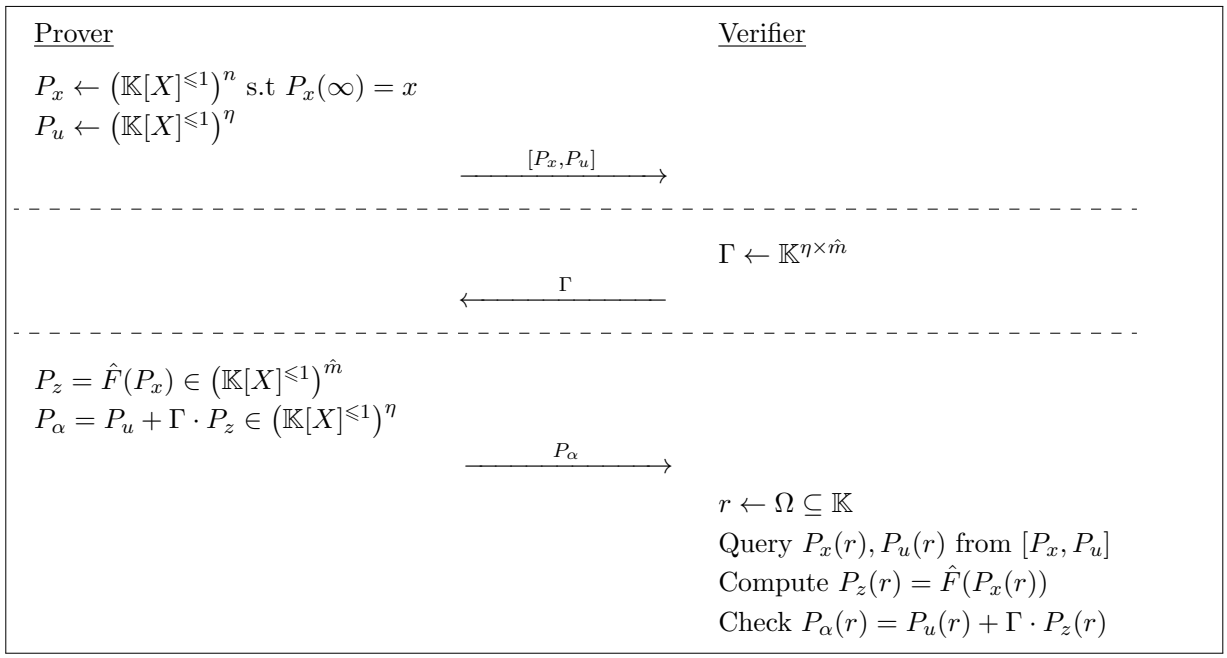


Figure 2: Polynomial IOP underlying MQOM (full version).

Remark 2. *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 packing strategy which significantly reduces the communication overhead in the context. Such an option was available in MQOM v2, but it was removed in MQOM v3 (see Section 6 for further discussion).*

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 malicious 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

$$\begin{cases} \texttt{node[1]} \leftarrow \texttt{rseed} \\ (\texttt{node[}2i\texttt{]}, \texttt{node[}2i+1\texttt{]}) \leftarrow \mathsf{SeedDerive}(\texttt{node[}i\texttt{]}) \quad \text{for } 1 \leq i \leq N-1 \end{cases} \tag{3}$$

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.

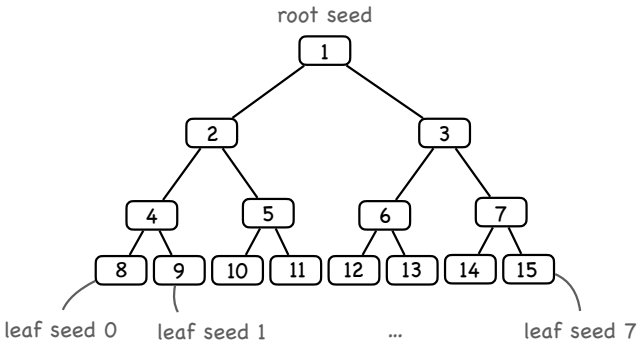


Figure 3: GGM tree structure and node numbering.

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:

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

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 3 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 $\texttt{rseed}$, one expands a GGM seed tree to obtain the leaf seeds $\texttt{lseed}[0]$, ..., $\texttt{lseed}[N-1]$. Then, each leaf seed is expanded into a share:

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

and a correction value Δ_x is defined as:

$$\Delta_x = x - \sum_{i=0}^{N-1} \bar{x}_i \ . \tag{4}$$

By definition, $(\bar{x}_0 + \Delta_x)$, $\bar{x}_1$, ..., $\bar{x}_{N-1}$ 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 $\texttt{ls\_com}[i^*]$ 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:

$$P_x = \Delta_x P_0 + \sum_{i=0}^{N-1} \bar{x}_i P_i \ \in \mathbb{K}[X]^{\leqslant 1} \tag{5}$$

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

$$\begin{cases} P_i(\omega_i) = 0 \\ P_i(\infty) = 1 \end{cases} \tag{6}$$

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 5 and Equation 6, 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 3. *As mentioned 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 6.*

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:

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

Then redefining:

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

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 bits of x, Equation 4 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.

This optimization is used in the correlated-tree variant of MQOM.

One tree optimization. Another optimization technique was proposed in [BBM+24] to slightly reduce the communication cost. Since the zero-knowledge protocol must be repeated τ times to achieve the desired security level (see the next section), the prover would normally need to manage τ independent GGM trees. Instead, [BBM+24] proposes replacing these τ GGM trees of N leaves with a single big GGM tree containing τN leaves. In this construction, the i-th leaf seed of the e-th protocol repetition is associated with the $(\tau i + e)$-th leaf of the big tree.

Opening all but τ leaves of this big tree is more efficient than opening all but one leaf in each of the τ independent trees. Indeed, the sensitive paths corresponding to the hidden leaves tend to merge with high probability, thereby reducing the number of internal nodes that must be revealed.

Furthermore, when the resulting sibling path is still too large, [BBM+24] proposes to resample the opening challenge. More precisely, if the sampled opening challenge produces a sibling path containing more than a predefined threshold T_{open} of revealed nodes, the prover rejects the challenge and samples a new one by incrementing a nonce input to the hash computation (this nonce is shared with the grinding mechanism explained below). This rejection-sampling strategy bounds the size of the sibling path at the expense of a moderate computational overhead during challenge generation.

This optimization is used in the one-tree variant of MQOM.

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. 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 repetitions; 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.

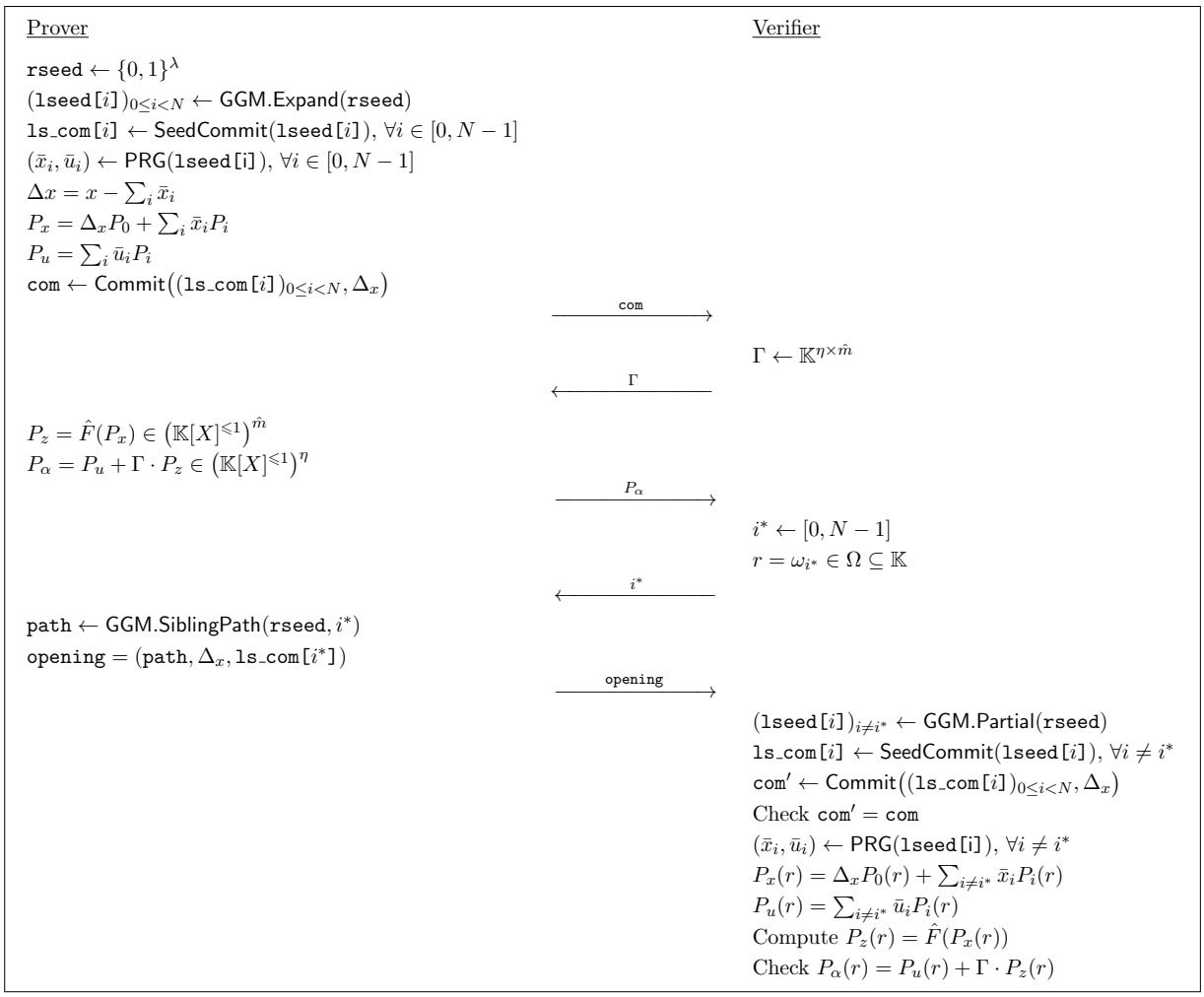


Figure 4: MQOM zero-knowledge proof of knowledge.

Parallel repetitions. The MQOM ZK PoK (Figure 4) has round-by-round soundness with soundness error $\epsilon_1 = 1/|\mathbb{K}|^\eta$ for the 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 repetitions. 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 $P_\alpha^{(e)} = P_u^{(e)} + \Gamma^{(e)} \cdot \hat{F}(P_x^{(e)})$ is computed and sent to the verifier, where the matrix $\Gamma^{(e)}$ differs for each repetition. The verifier then picks τ indexes $i^*[0], \ldots, i^*[\tau-1]$, each leading to an evaluation point $r^{(e)} \in \Omega$. The prover finally reveals 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 $(\texttt{lseed}[e][i])_{0\le i<N}$, corresponding commitments $(\texttt{ls\_com}[e][i])_{0\le i<N}$, and correction value $\Delta_x[e]$, for $e \in [0, \tau-1]$. The per-repetition global commitment is then defined

as:

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

It leads to τ commitment digests, no global commitment are computed. Finally, the global opening consists of the opening tuple $(\mathtt{path}[e], \Delta_x[e], \mathtt{ls\_com}[e][i^*[e]])$ for each execution $e \in [0, \tau-1]$.

Remark 4. *When using the one-tree technique, the τ parallel GGM trees are merged into a single big tree. The resulting τN leaf seeds are mapped to the values $(\mathtt{lseed}[e][i])_{0\leq e<\tau,\, 0\leq i<N}$, where the leaf corresponding to the i-th seed of the e-th execution is assigned to position $\tau i + e$ in the big GGM tree.*

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 $P_\alpha^{(e)}(r^{(e)})$ and computes a candidate value for the constant term:

$$\alpha_0^{(e)} = P_\alpha^{(e)}(r^{(e)}) - \alpha_1^{(e)} \cdot r^{(e)} \ .$$

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 $\mathtt{com}_1[0], \ldots, \mathtt{com}_1[\tau-1]$ the per-repetition BLC commitments 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 round, the application of Fiat-Shamir consists in deriving for each repetition a matrix $\Gamma[e] \in \mathbb{K}^{\eta \times \frac{m}{\mu}}$ from the corresponding commitment $\mathtt{com}_1[e]$ by a call to an extendable output hash function:

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

This produces τ independent batching challenges. Such an approach is sound because the batching soundness error of each repetition is already negligible. Consequently, deriving independent batching challenges does not introduce any security loss (see the paragraph "Independent challenges" in Section 6 for further discussion).

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 in this hash computation, which gives:

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

The inner hash is introduced to reduce the amount of precomputed data that must be stored when using pre-signatures (see Section 4.3).

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

$$\left(\frac{2}{N}\right)^\tau \leq \frac{1}{2^{\lambda - w}} \ , \tag{7}$$

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

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

and then iterates

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

where $i^* = (i^*[0], \ldots, i^*[\tau - 1]) \in [0, N-1]^\tau$, $\texttt{val} \in [0, 2^w - 1]$, and $\texttt{nonce} \in [0, 2^{32} - 1]$ a 32-bit counter. The above computation is repeated by increasing $\texttt{nonce}$ until obtaining $\texttt{val} = 0$. The succeeding $\texttt{nonce}$ 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 $\texttt{val} = 0$ 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 $\texttt{val} = 0$. 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 7. The reader is referred to [Sta21] for a formal argument.

In MQOM v3, we replace the hash-based proof-of-work described above with the block-cipher-based construction proposed by [Riv26]. The computation proceeds as follows:

$$\begin{aligned}
&(g_0, g_1, k_0, k_1) \leftarrow \mathsf{XOF}(\texttt{hash}) \\
&c_0 \leftarrow \mathsf{Enc}_{k_0}(g_0 \parallel \texttt{nonce}) \\
&c_1 \leftarrow \mathsf{Enc}_{k_1}(g_1 \parallel \texttt{nonce}) \\
&\texttt{val} \leftarrow \mathsf{Truncated}_{w-1}(c_0 \oplus c_1) \\
&i^* \leftarrow \mathsf{XOF}(\texttt{hash}, \texttt{nonce}, c_0, c_1)
\end{aligned}$$

The above procedure is repeated with increasing values of $\texttt{nonce}$ until $\texttt{val} = 0$. In the ideal cipher model (ICM), any attempt to forge a signature requires the adversary to perform the two corresponding cipher queries in order to derive the associated challenge i^*. Due to the grinding mechanism, each such pair of queries yields a valid grinding value $\texttt{val} = 0$ with probability $2^{-(w-1)}$. Consequently, the expected number of cipher queries required to obtain a valid challenge is approximately 2^w. The reader is referred to [Riv26] for a formal security analysis.

Seed derivation and commitment in GGM trees. The functions $\mathsf{SeedDerive}$, $\mathsf{SeedCommit}$, and $\mathsf{SeedExpand}$ are defined based on a block cipher:

$$\mathsf{Enc} : (\texttt{key}, \texttt{ptx}) \in \{0,1\}^\lambda \times \{0,1\}^\lambda \mapsto \texttt{ctx} \in \{0,1\}^\lambda \ , \tag{8}$$

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:

$$\psi : (x_l \parallel x_r) \in \{0,1\}^\lambda \mapsto (x_l \oplus x_r \parallel x_l) \ . \tag{9}$$

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

$$\mathsf{EncFF} : (\mathsf{seed}, \mathtt{tweak}) \mapsto \mathsf{Enc}(\mathtt{salt} \parallel \mathtt{tweak}, \mathsf{seed}) \oplus \psi(\mathsf{seed}) \ . \tag{10}$$

This allows us to define the seed derivation function as:

$$\mathsf{SeedDerive}(\mathsf{seed}, \mathtt{tweak}) = \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak}) \ , \tag{11}$$

and the seed commitment function as:

$$\mathsf{SeedCommit}(\mathsf{seed}, \mathtt{tweak}) = \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak}) \parallel \mathsf{EncFF}(\mathsf{seed}, \mathtt{tweak} \oplus 1) \ . \tag{12}$$

The domain-separator tweaks are defined to ensure security. For seed derivation, we assign a unique 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 $\mathbb{K}$
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 let ψ be the $\mathbb{F}_2$-linear orthomorphism defined as:

$$\psi : (x_l \parallel x_r) \in \{0,1\}^\lambda \mapsto (x_l \oplus x_r \parallel x_l) \ . \tag{13}$$

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 $\mu = [\mathbb{K} : \mathbb{F}]$
$\hat{m}$	Number of packed equations $\hat{m} = m/\mu$
N	Size of the evaluation domain $N = \lvert\Omega\rvert$
η	Number of internal repetitions $\eta = \lambda / \log_2 \lvert\mathbb{K}\rvert$
τ	Number of external repetitions of the proof system
h_B	Height of the large tree $h_B = \lceil \log_2(\tau \cdot N) \rceil$
w	Grinding proof-of-work parameter
T_{open}	Size (in seeds) of the sibling path when using the one-tree technique

Table 3: Notations of the MQOM signature scheme.

x	Secret MQ solution	$\mathbb{F}^n$
$\hat{A}_i$	Quadratic-part matrix of the i-th MQ packed equation	$\mathbb{K}^{n\times n}$
$\hat{b}_i$	Linear-part vector of the i-th MQ packed equation	$\mathbb{K}^n$
$\hat{y}_i$	Constant part of the i-th MQ packed equation	$\mathbb{K}$
`mseed_eq`	Global seed for the MQ packed equations $\{\hat{A}_i\}$, $\{\hat{b}_i\}$	$\{0,1\}^{2\lambda}$
`seed_eq`$[i-1]$	Seed for the i-th MQ packed equation $\hat{A}_i$, $\hat{b}_i$	$\{0,1\}^{\lambda}$
`com`$_1[e]$	BLC commitment of the e-th parallel repetition	$\{0,1\}^{2\lambda}$
`com`$_2$	Hash commitment of the polynomial P_α	$\{0,1\}^{2\lambda}$
`presig_id`	Pre-signature identifier	$\{0,1\}^{2\lambda}$
`sig_id`	Signature identifier	$\{0,1\}^{2\lambda}$
`key`	BLC opening key (variant-dependent structure)	–
`opening`	BLC opening (variant-dependent structure)	–
i^*	Hidden-leaf indexes $i^* = (i^*[0], \ldots, i^*[\tau-1])$	$[0, N-1]^\tau$
`nonce`	Grinding nonce	$\{0,1\}^{32}$
Δ_x	Auxiliary values $\Delta_x = (\Delta_x[0], \ldots, \Delta_x[\tau-1])$ for x	$(\mathbb{F}^n)^\tau$
$\Delta_x^{(1)}$	Partial auxiliary values $\Delta_x^{(1)} = (\Delta_x^{(1)}[0], \ldots, \Delta_x^{(1)}[\tau-1])$	$\left(\{0,1\}^{\lvert x\rvert_2-\lambda}\right)^\tau$
x_0	Coefficient array $x_0 = (x_0[0], \ldots, x_0[\tau-1])$	$\left(\mathbb{K}^n\right)^\tau$
z_0	Coefficient array $z_0 = (z_0[0], \ldots, z_0[\tau-1])$	$\left(\mathbb{K}^{\hat{m}}\right)^\tau$
z_1	Coefficient array $z_1 = (z_1[0], \ldots, z_1[\tau-1])$	$\left(\mathbb{K}^{\hat{m}}\right)^\tau$
$\Gamma[e]$	Batching matrix of the e-th parallel repetition	$\mathbb{K}^{\eta\times\hat{m}}$
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 `x_eval` $= (\texttt{x\_eval}[0], \ldots, \texttt{x\_eval}[\tau-1])$	$(\mathbb{K}^n)^\tau$
`u_eval`	Evaluation array `u_eval` $= (\texttt{u\_eval}[0], \ldots, \texttt{u\_eval}[\tau-1])$	$(\mathbb{K}^{\eta})^\tau$
r	Evaluation point $\omega_{i^*[e]}$ of the current parallel repetition	$\Omega \subset \mathbb{K}$
v_z	Evaluation of P_α at r for the current parallel repetition	$\mathbb{K}^{\hat{m}}$
`mseed`	Signature master seed	$\{0,1\}^{\lambda}$
`salt`	Signature salt	$\{0,1\}^{\lambda}$
`tree`	GGM trees (variant-dependent structure)	–
`lseed`	Array `lseed` $= (\texttt{lseed}[0], \ldots, \texttt{lseed}[\tau-1])$	–
`lseed`$[e]$	Array `lseed`$[e] = (\texttt{lseed}[e][0], \ldots, \texttt{lseed}[e][N-1])$	–
`lseed`$[e][i]$	Leaf seed of index i for execution e	$\{0,1\}^{\lambda}$
`lseed_all`	Array $(\texttt{lseed}[0][0], \texttt{lseed}[1][0], \ldots, \texttt{lseed}[\tau-1][N-1])$	$\left(\{0,1\}^{\lambda}\right)^{\tau\cdot N}$
`ls_com`	Array `ls_com` $= (\texttt{ls\_com}[0], \ldots, \texttt{ls\_com}[\tau-1])$	–
`ls_com`$[e]$	Array `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	$\{0,1\}^{2\lambda}$
`out_ls_com`	Array `out_ls_com` $= (\texttt{out\_ls\_com}[0], \ldots, \texttt{out\_ls\_com}[\tau-1])$	–
`out_ls_com`$[e]$	Seed commitment of the e-th hidden leaf seed	$\{0,1\}^{2\lambda}$
`path`	Sibling path in the GGM trees (variant-dependent structure)	–

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 performs 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 $\lceil \ell/8 \rceil$. More precisely, a bitstring $s := (s_1, \ldots, s_\ell) \in \{0,1\}^\ell$, for which the size is a multiple of 8, is represented as:

$$\mathsf{B}(s_1, \ldots, s_8) \parallel \mathsf{B}(s_9, \ldots, s_{16}) \parallel \ldots \parallel \mathsf{B}(s_{\ell-7}, \ldots, s_\ell)$$

where B is the byte-grouping function defined as:

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

By design, ℓ is always a multiple of 8 in the specification.

The different variants of MQOM involve three different fields: $\mathbb{F}_2$, $\mathbb{F}_{16}$, $\mathbb{F}_{256}$ 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 $\mathbb{F}_{16}$-elements. An element of $\mathbb{F}_{16}$ is naturally represented as a half byte, commonly referred to as a *nibble*. Specifically, an element $e \in \mathbb{F}_{16}$ is represented as a tuple $(e_0, \ldots, e_3) \in \mathbb{F}_2^4$ such that $e = \sum_{i=0}^{3} e_i \cdot \rho^i$ for ρ a primitive element of $\mathbb{F}_{16}$ over $\mathbb{F}_2$ (i.e. $\mathbb{F}_{16} = \mathbb{F}_2[\rho]$). The byte representation of a pair $(e_1, e_2) \in \mathbb{F}_{16}^2$ is defined as:

$$\mathsf{byte}(e_1, e_2) = \mathsf{B}(e_{1,0}, e_{1,1}, e_{1,2}, e_{1,3},\ e_{2,0}, e_{2,1}, e_{2,2}, e_{2,3}) \quad \Leftrightarrow \quad e_i = \sum_{j=0}^{3} e_{i,j} \cdot \rho^j \ ,\ \forall i \in \{1,2\} \ .$$

Vectors from $\mathbb{F}_{16}^\ell$ are only serialized for ℓ a multiple of 2. A vector $v = (v_1, \ldots, v_\ell) \in \mathbb{F}_{16}^\ell$ is naturally serialized as:

$$\mathsf{Serialize}(v) = \mathsf{byte}(v_1, v_2) \parallel \ldots \parallel \mathsf{byte}(v_{\ell-1}, v_\ell) \ .$$

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

Tuples, arrays and matrices. 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 $\mathtt{arr} = (\mathtt{arr[0]}, \ldots, \mathtt{arr}[\ell - 1])$ is serialized as

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

Matrices of elements are serialized in a row-major fashion: a matrix M of size $n \times m$,

$$M = \begin{pmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,m} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n,1} & e_{n,2} & \cdots & e_{n,m} \end{pmatrix}$$

is serialized as:

$$\mathsf{Serialize}(M) = \mathsf{Serialize}((e_{1,1}, \ldots, e_{1,m})) \parallel \ldots \parallel \mathsf{Serialize}((e_{n,1}, \ldots, e_{n,m}))$$

2.4 Main algorithms

2.4.1 Key generation

The key generation of MQOM consists in pseudorandomly generating a (packed) MQ instance, with triangular matrices $\hat{A}_i$. It randomly draws $\texttt{bitstr}_x$ from which the secret MQ solution x is derived, and another seed $\texttt{mseed\_eq}$ from which the packed MQ equations $\{\hat{A}_i\}, \{\hat{b}_i\}$ are derived. The MQ output $\hat{y}$ is then computed from $\{\hat{A}_i\}, \{\hat{b}_i\}$ and x. Finally, the key pair is defined and returned as $\texttt{pk} := (\texttt{mseed\_eq}, \hat{y})$ and $\texttt{sk} := (\texttt{pk}, x)$. The key generation is depicted in Algorithm 1. The subroutine ExpandEquations is invoked to expand the MQ equations from the seed $\texttt{mseed\_eq}$. Subseeds $\texttt{seed\_eq[0]}, \ldots, \texttt{seed\_eq}[\hat{m}-1]$ are first derived from $\texttt{mseed\_eq}$, then $(\hat{A}_i, \hat{b}_i)$ is expanded from $\texttt{seed\_eq}[i]$ for every $i \in [1, \hat{m}]$.

Algorithm 1 KeyGen()

Output: a public key $\texttt{pk}$, a secret key $\texttt{sk}$

1: $(\texttt{bitstr}_x, \texttt{mseed\_eq}) \leftarrow \{0,1\}^{n \cdot \log_2 |\mathbb{F}|} \times \{0,1\}^{2\lambda}$
2: $x \leftarrow \mathsf{Parse}(\texttt{bitstr}_x)$ ▷ $x \in \mathbb{F}^n$
3: $(\{\hat{A}_i\}, \{\hat{b}_i\}) \leftarrow \mathsf{ExpandEquations}(\texttt{mseed\_eq})$ ▷ $\hat{A}_i \in \mathbb{K}^{n \times n}, \hat{b}_i \in \mathbb{K}^n$
4: **for** $i = 1$ **to** $\hat{m}$ **do**
5: $\quad \hat{y}_i \leftarrow x^\intercal \hat{A}_i x + \hat{b}_i^\intercal x$ ▷ $\hat{y}_i \in \mathbb{K}$
6: $\hat{y} \leftarrow (\hat{y}_1, \ldots, \hat{y}_{\hat{m}})$ ▷ $\hat{y} \in \mathbb{K}^{\hat{m}}$
7: $\texttt{pk} \leftarrow \mathsf{Serialize}(\texttt{mseed\_eq}, \hat{y})$
8: $\texttt{sk} \leftarrow \mathsf{Serialize}(\texttt{pk}, x)$
9: **return** $(\texttt{pk}, \texttt{sk})$

Algorithm 2 ExpandEquations($\texttt{mseed\_eq}$)

Input: a seed key $\texttt{mseed\_eq} \in \{0,1\}^{2\lambda}$
Output: Packed MQ equations $(\{\hat{A}_i\}, \{\hat{b}_i\})$

1: Let $nf_{\text{eq}} = n + \sum_{j=1}^{n} j$ ▷ Number of field elements for $(\hat{A}_i, \hat{b}_i)$
2: Let $nb_{\text{eq}} = nf_{\text{eq}} \cdot \frac{\log_2 |\mathbb{K}|}{8}$ ▷ Number of PRG bytes for $(\hat{A}_i, \hat{b}_i)$
3: **for** $i = 1$ **to** $\hat{m}$ **do**
4: $\quad \texttt{seed\_eq}[i-1] \leftarrow \mathsf{XOF}_0((\texttt{mseed\_eq}, \mathsf{Bits}_{16}(i-1)), \texttt{len} := \lambda)$
5: $\quad (\texttt{row}_{i,1}, \ldots, \texttt{row}_{i,n}, \hat{b}_i) \leftarrow \mathsf{Parse}(\mathsf{PRG}(\texttt{seed\_eq}[i-1], nb_{\text{eq}}))$ ▷ $\text{row}_{i,j} \in \mathbb{K}^j, \hat{b}_i \in \mathbb{K}^n$
6: $\quad$ **for** $j = 1$ **to** n **do**
7: $\quad\quad \hat{A}_{i,j} \leftarrow (\texttt{row}_{i,j} \parallel 0_{\mathbb{K}^{n-j}})$ ▷ $\hat{A}_{i,j} \in \mathbb{K}^n$, jth row of $\hat{A}_i$
8: $\quad \hat{A}_i \leftarrow (\hat{A}_{i,1}, \ldots, \hat{A}_{i,n})$ ▷ $\hat{A}_i \in \mathbb{K}^{n \times n}$
9: **return** $(\{\hat{A}_i\}, \{\hat{b}_i\})$

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 10. The subroutines for the BLC commitment and computation of P_α are depicted in Section 2.5. The signing procedure may fail if the grinding procedure within SampleChallenge does not find a valid challenge. However, with the parameters used in MQOM, the failure probability is smaller than 2^{-1000} for all instances. A discussion of the worst-case execution time is provided in Section 4.5.

Algorithm 3 Sign(sk, msg)

Input: a secret key sk, a message msg
Output: a signature sig

1: $(\mathtt{pk}, x) = \mathsf{Parse}(\mathtt{sk})$
2: $(\mathtt{mseed\_eq}, \hat{y}) = \mathsf{Parse}(\mathtt{pk})$
3: $\mathtt{mseed} \leftarrow \{0,1\}^\lambda$
4: $\mathtt{salt} \leftarrow \{0,1\}^\lambda$
5: $(\mathtt{com}_1, \mathtt{key}, x_0, u_0, u_1) \leftarrow \mathsf{BLC.Commit}(\mathtt{mseed}, \mathtt{salt}, x)$
6: $(\alpha_0, \alpha_1) \leftarrow \mathsf{ComputePAlpha}(\mathtt{com}_1, x_0, u_0, u_1, x, \mathtt{mseed\_eq})$
7: $\mathtt{com}_2 \leftarrow \mathsf{Hash}_1(\{(\alpha_0[e], \alpha_1[e])\}_e)$ ▷ $\mathsf{Hash}_1(\alpha_0[0], \alpha_1[0], \alpha_0[1], \ldots)$
8: $\mathtt{presig\_id} \leftarrow \mathsf{Hash}_2(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2)$
9: $\mathtt{msg\_hash} \leftarrow \mathsf{Hash}_3(\mathtt{msg})$
10: $\mathtt{sig\_id} \leftarrow \mathsf{Hash}_4(\mathtt{presig\_id}, \mathtt{msg\_hash})$
11: $(i^*, \mathtt{nonce}) \leftarrow \mathsf{SampleChallenge}(\mathtt{sig\_id})$
12: **if** $(i^*, \mathtt{nonce}) = \perp$, **return** $\perp$
13: $\mathtt{opening} \leftarrow \mathsf{BLC.Open}(\mathtt{key}, i^*, \alpha_1)$
14: **return** sig := $\mathsf{Serialize}(\mathtt{sig\_id}, \mathtt{salt}, \mathtt{nonce}, \mathtt{opening})$

2.4.3 Verification

The MQOM verification process is depicted in Algorithm 4. The subroutines for the BLC evaluation and recomputation of P_α are depicted in Section 2.5.

Algorithm 4 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: $(\mathtt{mseed\_eq}, \hat{y}) = \mathsf{Parse}(\mathtt{pk})$
2: $(\mathtt{sig\_id}, \mathtt{salt}, \mathtt{nonce}, \mathtt{opening}) = \mathsf{Parse}(\mathsf{sig})$
3: $i^* \leftarrow \mathsf{DeriveChallenge}(\mathtt{sig\_id}, \mathtt{nonce})$
4: **if** $i^* = \perp$ **then return** False
5: $(\mathtt{com}_1, \mathtt{x\_eval}, \mathtt{u\_eval}, \alpha_1) \leftarrow \mathsf{BLC.Eval}(\mathtt{salt}, \mathtt{opening}, i^*)$
6: $\alpha_0 \leftarrow \mathsf{RecomputePAlpha}(\mathtt{com}_1, \alpha_1, i^*, \mathtt{x\_eval}, \mathtt{u\_eval}, \mathtt{mseed\_eq}, \hat{y})$
7: $\mathtt{com}_2 \leftarrow \mathsf{Hash}_1(\{(\alpha_0[e], \alpha_1[e])\}_e)$
8: $\mathtt{presig\_id} \leftarrow \mathsf{Hash}_2(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2)$
9: $\mathtt{msg\_hash} \leftarrow \mathsf{Hash}_3(\mathtt{msg})$
10: $\mathtt{sig\_id}' \leftarrow \mathsf{Hash}_4(\mathtt{presig\_id}, \mathtt{msg\_hash})$
11: **if** $\mathtt{sig\_id}' \neq \mathtt{sig\_id}$ **then return** False
12: **return** True

2.5 Subroutines

In what follows, we describe all the subroutines used in MQOM. Some of these routines are variant-specific. Whenever such a routine, denoted RoutineName, is invoked, its correlated-tree instantiation is written as CT.RoutineName, whereas its one-tree instantiation is denoted by OT.RoutineName. The variant-specific routines are listed in Table 4.

Routine	Correlated Trees	One big Tree
BLC.Commit	CT.BLC.Commit	OT.BLC.Commit
BLC.IsValidChallenge	CT.BLC.IsValidChallenge	OT.BLC.IsValidChallenge
BLC.Open	CT.BLC.Open	OT.BLC.Open
BLC.Eval	CT.BLC.Eval	OT.BLC.Eval
SeedExpand	CT.SeedExpand	OT.SeedExpand

Table 4: List of the variant-specific subroutines

2.5.1 Arithmetic routines

The main arithmetic routine of the signing process is ComputePAlpha which computes the polynomials $P_\alpha = P_u + \hat{F}(P_x)$, for all the executions $e \in [0, \tau - 1]$, where $\hat{F} = (\hat{f}_1, \ldots, \hat{f}_{\hat{m}})$ with $\hat{f}_i(x) = x^\intercal \hat{A}_i x + \hat{b}_i^\intercal x - \hat{y}_i$. 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 $P_z = \hat{F}(P_x)$ from P_x. The batching (a.k.a. 5-round) variant is enabled/disabled with the Boolean `batching`.

Algorithm 5 ComputePAlpha($\mathtt{com}_1, x_0, u_0, u_1, x, \mathtt{mseed\_eq}$)

Input: a BLC commitment $\mathtt{com}_1$, coefficient arrays x_0, u_0, u_1, MQ secret solution x, seed $\mathtt{mseed\_eq}$ of the packed MQ equations $\{\hat{A}_i\}, \{\hat{b}_i\}$
Output: coefficient arrays α_0, α_1

1: $(\{\hat{A}_i\}, \{\hat{b}_i\}) \leftarrow$ ExpandEquations($\mathtt{mseed\_eq}$)
2: **for** $e = 0$ **to** $\tau - 1$ **do**
3: $\quad (z_0, z_1) \leftarrow$ ComputePz($x_0[e], x, \{\hat{A}_i\}, \{\hat{b}_i\}$) ▷ $P_z = z_0 + z_1 \cdot X \in \left(\mathbb{K}[X]^{\leq 1}\right)^{\hat{m}}$
4: $\quad \Gamma[e] \leftarrow$ Parse(XOF$_8$($\mathtt{com}_1[e], \mathtt{len} := \eta \cdot \hat{m} \cdot \log_2 |\mathbb{K}|$)) ▷ $\Gamma[e] \in \mathbb{K}^{\eta \times \hat{m}}$
5: $\quad \alpha_0[e] \leftarrow u_0[e] + \Gamma[e] \cdot z_0$
6: $\quad \alpha_1[e] \leftarrow u_1[e] + \Gamma[e] \cdot z_1$ ▷ $P_\alpha = \alpha_0[e] + \alpha_1[e] \cdot X \in \left(\mathbb{K}[X]^{\leq 1}\right)^{\eta}$
7: **return** (α_0, α_1)

Algorithm 6 ComputePz($x_0[e], x, \{\hat{A}_i\}, \{\hat{b}_i\}$)

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

▷ Compute $P_{z,i} = P_x{}^\intercal \hat{A}_i P_x + \hat{b}_i^\intercal P_x \cdot X - y_i \cdot X^2$ for all $i \in [1, \hat{m}]$
▷ Skip computation of degree-2 coefficients (known to be 0)

1: **for** $i = 1$ **to** $\hat{m}$ **do**
▷ Compute $P_t = t_0 + t_1 \cdot X := \hat{A}_i \cdot P_x + \hat{b}_i \cdot X$
2: $\quad t_0 \leftarrow \hat{A}_i \cdot x_0[e]$ ▷ $t_0 \in \mathbb{K}^n$
3: $\quad t_1 \leftarrow \hat{A}_i \cdot x + \hat{b}_i$ ▷ $t_1 \in \mathbb{K}^n$
▷ Compute $P_{z,i} = z_{0,i} + z_{1,i} \cdot X = P_t{}^\intercal P_x - y_i \cdot X^2$
4: $\quad z_{0,i} \leftarrow t_0{}^\intercal \cdot x_0[e]$ ▷ $z_{0,i} \in \mathbb{K}$
5: $\quad z_{1,i} \leftarrow t_0{}^\intercal \cdot x + t_1{}^\intercal \cdot x_0[e]$ ▷ $z_{1,i} \in \mathbb{K}$
6: $z_0 \leftarrow (z_{0,1}, \ldots, z_{0,\hat{m}})$ ▷ $z_0 \in \mathbb{K}^{\hat{m}}$
7: $z_1 \leftarrow (z_{1,1}, \ldots, z_{1,\hat{m}})$ ▷ $z_1 \in \mathbb{K}^{\hat{m}}$
8: **return** (z_0, z_1)

Remark 5. *In* ComputePz*, the variable* $t_1 = \hat{A}_i \cdot x + \hat{b}_i$ *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* $\hat{m}$ *values of* t_1 *(corresponding to* $i \in [1, \hat{m}]$*) 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 7 RecomputePAlpha($\mathtt{com}_1, \alpha_1, i^*, \mathtt{x\_eval}, \mathtt{u\_eval}, \mathtt{mseed\_eq}, \{\hat{y}_i\}$)

Input: a BLC commitment $\mathtt{com}_1$, coefficient array α_1, index array i^*, evaluation arrays $\mathtt{x\_eval}$, $\mathtt{u\_eval}$, seed $\mathtt{mseed\_eq}$ of the packed MQ equations $\{\hat{A}_i\}, \{\hat{b}_i\}$, and $\{\hat{y}_i\}$
Output: coefficient array α_0

1: $(\{\hat{A}_i\}, \{\hat{b}_i\}) \leftarrow$ ExpandEquations($\mathtt{mseed\_eq}$)
2: **for** $e = 0$ **to** $\tau - 1$ **do**
3: Let $r = \omega_{i^*[e]}$ ▷ $r \in \mathbb{K}$
4: Let $v_x = \mathtt{x\_eval}[e]$ ▷ $v_x = P_x(r) \in \mathbb{K}^n$
5: Let $v_u = \mathtt{u\_eval}[e]$ ▷ $v_u = P_u(r) \in \mathbb{K}^\eta$
6: $v_z \leftarrow$ ComputePzEval$(r, v_x, \{\hat{A}_i\}, \{\hat{b}_i\}, \{\hat{y}_i\})$ ▷ $v_z = P_z(r) \in \mathbb{K}^{\hat{m}}$
7: $\Gamma[e] \leftarrow$ Parse(XOF$_8(\mathtt{com}_1[e], \mathtt{len} := \eta \cdot \hat{m} \cdot \log_2 |\mathbb{K}|)$) ▷ $\Gamma[e] \in \mathbb{K}^{\eta \times \hat{m}}$
8: $v_\alpha \leftarrow v_u + \Gamma[e] \cdot v_z$ ▷ $v_\alpha = P_\alpha(r) = \alpha_0[e] + \alpha_1[e] \cdot r$
9: $\alpha_0[e] \leftarrow v_\alpha - \alpha_1[e] \cdot r$ ▷ $\alpha_0[e] \in \mathbb{K}^\eta$
10: **return** α_0

Algorithm 8 ComputePzEval$(r, v_x, \{\hat{A}_i\}, \{\hat{b}_i\}, \{\hat{y}_i\})$

Input: evaluation point $r \in \Omega$, evaluation $v_x \in \mathbb{K}$, MQ equations $\{\hat{A}_i\}, \{\hat{b}_i\}, \{\hat{y}_i\}$
Output: evaluation $v_z \in \mathbb{K}^{\hat{m}}$

▷ Compute $v_{z,i} = v_x{}^\intercal \hat{A}_i v_x + \hat{b}_i^\intercal v_x \cdot r - y_i \cdot r^2$ for all $i \in [1, \hat{m}]$
1: **for** $i = 1$ **to** $\hat{m}$ **do**
▷ Compute $v_t = P_t(r) = \hat{A}_i \cdot P_x(r) + \hat{b}_i \cdot r$
2: $v_t \leftarrow \hat{A}_i \cdot v_x + \hat{b}_i \cdot r$ ▷ $v_t \in \mathbb{K}^n$
▷ Compute $v_{z,i} = P_{z,i}(r) = v_t^\intercal v_x - \hat{y}_i \cdot r^2$
3: $v_{z,i} \leftarrow v_t^\intercal \cdot v_x - \hat{y}_i \cdot r^2$ ▷ $v_{z,i} \in \mathbb{K}$
4: $v_z \leftarrow (v_{z,1}, \ldots, v_{z,\hat{m}})$ ▷ $v_z \in \mathbb{K}^{\hat{m}}$
5: **return** v_z

2.5.2 Grinding procedure

The grinding procedure is implemented through two routines. The routine DeriveChallenge derives an opening challenge from a signature identifier and a nonce. It may return $\perp$ if either the derived challenge or the corresponding grinding value is invalid. The routine SampleChallenge then searches for a valid opening challenge by repeatedly invoking DeriveChallenge with different nonce values.

Remark 6. *Several computations performed by* DeriveChallenge *are independent of the nonce. When* DeriveChallenge *is called from* SampleChallenge*, these computations can therefore be precomputed once before entering the search loop.*

Algorithm 9 DeriveChallenge($\texttt{sig\_id}, \texttt{nonce}$)

Input: Signature identifier $\texttt{sig\_id} \in \{0,1\}^{2\lambda}$, a grinding nonce $\texttt{nonce} \in \{0,1\}^{32}$
Output: challenge indexes $i^* \in [0, N-1]^\tau \cup \{\perp\}$

▷ Computation independent on the nonce, can be precomputed
$(g_0, g_1, k_0, k_1) \leftarrow \mathsf{Parse}(\mathsf{XOF}_5(\texttt{sig\_id}, \texttt{len} := 4\lambda - 64))$ ▷ $g_0, g_1 \in \{0,1\}^{\lambda-32}$
▷ $k_0, k_1 \in \{0,1\}^{\lambda}$
▷ Computation dependent on the nonce
$c_0 \leftarrow \mathsf{Enc}(\texttt{key} := (\mathsf{Truncate}_{\lambda-2}(k_0) \parallel 01), \texttt{ptx} := (g_0 \parallel \texttt{nonce}))$
$c_1 \leftarrow \mathsf{Enc}(\texttt{key} := (\mathsf{Truncate}_{\lambda-2}(k_1) \parallel 11), \texttt{ptx} := (g_1 \parallel \texttt{nonce}))$
if $\mathsf{Truncate}_{w-1}(c_0 \oplus c_1) = 0$ **then**
  $\texttt{unred}_{i^*} \leftarrow \mathsf{Parse}(\mathsf{XOF}_6((\texttt{sig\_id}, \texttt{nonce}, c_0, c_1), \texttt{len} := 16\tau))$ ▷ $\texttt{unred}_{i^*} \in [0, 2^{16}-1]^\tau$
  $i^*[e] = \texttt{unred}_{i^*}[e] \bmod N$, for all e ▷ $i^* \in [0, N-1]^\tau$
  if $\mathsf{BLC.IsValidChallenge}(i^*) = \textbf{true}$ **then**
    return i^*
return $\perp$

Algorithm 10 SampleChallenge($\texttt{sig\_id}$)

Input: Signature identifier $\texttt{sig\_id} \in \{0,1\}^{2\lambda}$
Output: challenge indexes $i^* \in [0, N-1]^\tau$, a grinding nonce $\texttt{nonce} \in \{0,1\}^{32}$

for $\texttt{ctr} = 0$ **to** $2^{32} - 1$ **do**
  $\texttt{nonce} \leftarrow \mathsf{Bits}_{32}(\texttt{ctr})$
  $i^* \leftarrow \mathsf{DeriveChallenge}(\texttt{sig\_id}, \texttt{nonce})$
  if $i^* \neq \perp$ **then**
    return $(i^*, \texttt{nonce})$
return $\perp$ ▷ Failure to find a valid challenge

2.5.3 Batch line commitment routines

Line Conversion. The BLC.ConvertToLine routine transforms an array of N seeds into the line representations $P_x := x_0 + x \cdot X$ and $P_u := u_0 + u_1 \cdot X$. The BLC.ConvertToLineEvaluation routine takes as input an array of N seeds in which the i^*-th seed is missing, and computes the corresponding line evaluations $v_x := P_x(\omega_{i^*})$ and $v_u := P_u(\omega_{i^*})$.

Algorithm 11 BLC.ConvertToLine($\texttt{salt}, e, \texttt{lseed}[e], x$)

Input: a salt $\texttt{salt} \in \{0,1\}^\lambda$, an execution index $e \in [0, \tau-1]$, seed array $\texttt{lseed}[e] \in (\{0,1\}^\lambda)^N$, an MQ secret solution x
Output: seed commitments $\texttt{ls\_com}[e]$, an auxiliary value $\Delta_x[e] \in \mathbb{F}^n$, coefficients x_0, u_0, u_1

1: $\texttt{tweaked\_salt}_0 \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 0, e)$
2: $\texttt{tweaked\_salt}_1 \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 1, e)$
3: **for** $i = 0$ **to** $N-1$ **do**
4: $\quad \texttt{ls\_com}[e][i] \leftarrow \mathsf{SeedCommit}(\texttt{tweaked\_salt}_0, \texttt{tweaked\_salt}_1, \texttt{lseed}[e][i])$
5: $\quad (\bar{x}_i, \bar{u}_i) \leftarrow \mathsf{Parse}(\mathsf{SeedExpand}(\texttt{salt}, e, \texttt{lseed}[e][i], |x|_8 + |u|_8))$ ▷ $\bar{x}_i \in \mathbb{F}^n$, $\bar{u}_i \in \mathbb{K}^\eta$

The below operations can be efficiently implemented by taking advantage of the structure of $\{\omega_i\}_i$, see Section 4.1 for details.

▷ Compute $P_u = u_0 + u_1 \cdot X = \sum_{i=0}^{N-1} \bar{u}_i \cdot (X - \omega_i)$
6: $u_0 \leftarrow -\sum_{i=0}^{N-1} \omega_i \cdot \bar{u}_i$ ▷ $u_0 \in \mathbb{K}^\eta$
7: $u_1 \leftarrow \sum_{i=0}^{N-1} \bar{u}_i$ ▷ $u_1 \in \mathbb{K}^\eta$

▷ Compute $P_x = x_0 + x \cdot X = \Delta_x[e] \cdot X + \sum_{i=0}^{N-1} \bar{x}_i \cdot (X - \omega_i)$
8: $x_0 \leftarrow -\sum_{i=0}^{N-1} \omega_i \cdot \bar{x}_i$ ▷ $x_0 \in \mathbb{K}^n$
9: $\Delta_x[e] \leftarrow x - \sum_{i=0}^{N-1} \bar{x}_i$ ▷ $\Delta_x[e] \in \mathbb{F}^n$
10: **return** $(\texttt{ls\_com}[e], \Delta_x[e], x_0, u_0, u_1)$

Algorithm 12 BLC.ConvertToLineEvaluation($\texttt{salt}, e, \texttt{lseed}[e], \Delta_x[e], i^*$)

Input: a salt $\texttt{salt} \in \{0,1\}^\lambda$, an execution index $e \in [0, \tau-1]$, partial seed array $\texttt{lseed}[e]$, an auxiliary value $\Delta_x[e] \in \mathbb{F}^n$, an index i^*
Output: seed commitments $\texttt{ls\_com}[e]$, evaluations $(\texttt{x\_eval}[e], \texttt{u\_eval}[e])$

1: $\texttt{tweaked\_salt}_0 \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 0, e)$
2: $\texttt{tweaked\_salt}_1 \leftarrow \mathsf{TweakSalt}(\texttt{salt}, 1, e)$
3: **for** $i = 0$ **to** $N-1$ **do**
4: $\quad$ **if** $i \neq i^*$ **then**
5: $\quad\quad \texttt{ls\_com}[e][i] \leftarrow \mathsf{SeedCommit}(\texttt{tweaked\_salt}_0, \texttt{tweaked\_salt}_1, \texttt{lseed}[e][i])$
6: $\quad\quad (\bar{x}_i, \bar{u}_i) \leftarrow \mathsf{Parse}(\mathsf{SeedExpand}(\texttt{salt}, e, \texttt{lseed}[e][i], |x|_8 + |u|_8))$ ▷ $\bar{x}_i \in \mathbb{F}^n$, $\bar{u}_i \in \mathbb{K}^\eta$
7: Let $r = \omega_{i^*}$ ▷ $r \in \mathbb{K}$
8: $\texttt{x\_eval}[e] \leftarrow \Delta_x[e] \cdot r + \sum_{i \neq i^*} \bar{x}_i \cdot (r - \omega_i)$ ▷ $\texttt{x\_eval}[e] = P_x(r) \in \mathbb{K}^n$
9: $\texttt{u\_eval}[e] \leftarrow \sum_{i \neq i^*} \bar{u}_i \cdot (r - \omega_i)$ ▷ $\texttt{u\_eval}[e] = P_u(r) \in \mathbb{K}^\eta$
10: **return** $(\texttt{ls\_com}[e], \texttt{x\_eval}[e], \texttt{u\_eval}[e])$

Line Commitment. 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. Its implementation depends on the chosen variant: the correlated-tree instantiation is denoted by CT.BLC.Commit, while the one-tree instantiation is denoted by OT.BLC.Commit.

Algorithm 13 CT.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: BLC commitment array $\texttt{com}_1$, an opening key $\texttt{key}$, coefficient arrays x_0, u_0, u_1

$\delta \leftarrow \mathsf{FirstBits}_\lambda(x)$ ▷ $\delta \in \{0,1\}^\lambda$
for $e = 0$ **to** $\tau - 1$ **do**
  $(\texttt{tree}[e], \texttt{lseed}[e]) \leftarrow \mathsf{CT.SmallGGMTree.Expand}(\texttt{salt}, \texttt{mseed}, e, \delta)$
  $(\texttt{ls\_com}[e], \Delta_x[e], x_0[e], u_0[e], u_1[e]) \leftarrow \mathsf{BLC.ConvertToLine}(\texttt{salt}, e, \texttt{lseed}[e], x)$
  $\Delta_x^{(1)}[e] \leftarrow \mathsf{NextBits}_\lambda(\Delta_x[e])$ ▷ $\Delta_x^{(1)}[e] \in \{0,1\}^{|x|_2 - \lambda}$
  $\texttt{com}_1[e] \leftarrow \mathsf{Hash}_7(\mathsf{Bits}_8(e), \texttt{salt}, \texttt{ls\_com}[e], \Delta_x^{(1)}[e])$
$\texttt{key} \leftarrow \mathsf{Serialize}(\texttt{tree}, \texttt{ls\_com}, \Delta_x^{(1)})$
return $(\texttt{com}_1, \texttt{key}, x_0, u_0, u_1)$

Algorithm 14 OT.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

$(\texttt{tree}, \texttt{lseed\_all}) \leftarrow \mathsf{OT.BigGGMTree.Expand}(\texttt{salt}, \texttt{mseed})$
for $e = 0$ **to** $\tau - 1$ **do**
  for $i = 0$ **to** $N - 1$ **do**
    $\texttt{lseed}[e][i] = \texttt{lseed\_all}[\tau \cdot i + e]$
  $(\texttt{ls\_com}[e], \Delta_x[e], x_0[e], u_0[e], u_1[e]) \leftarrow \mathsf{BLC.ConvertToLine}(\texttt{salt}, e, \texttt{lseed}[e], x)$
  $\texttt{com}_1[e] \leftarrow \mathsf{Hash}_7(\mathsf{Bits}_8(e), \texttt{salt}, \texttt{ls\_com}[e], \Delta_x[e])$
$\texttt{key} \leftarrow \mathsf{Serialize}(\texttt{tree}, \texttt{ls\_com}, \Delta_x)$
return $(\texttt{com}_1, \texttt{key}, x_0, u_0, u_1)$

Challenge Verification. The BLC.IsValidChallenge routine returns a Boolean value indicating whether the given opening challenge is valid. When using correlated trees, an opening challenge is always valid.

Algorithm 15 CT.BLC.IsValidChallenge(i^*)

Input: index array i^*
Output: a Boolean value $\texttt{ret} \in \{\mathsf{True}, \mathsf{False}\}$

return True

Algorithm 16 OT.BLC.IsValidChallenge(i^*)

Input: index array i^*
Output: a Boolean value `ret` $\in \{\mathsf{True}, \mathsf{False}\}$

1: `hidden_leaf_idxs` $= (i^*[0] \cdot \tau + 0, \ldots, i^*[\tau - 1] \cdot \tau + (\tau - 1))$
2: **return** OT.BigGGMTree.IsValidOpeningSet(`hidden_leaf_idxs`)

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

Algorithm 17 CT.BLC.Open(`key`, i^*, α_1)

Input: a commitment key `key`, an index array i^*, τ vectors $\alpha_1[0], \ldots, \alpha_1[\tau - 1] \in \mathbb{K}^\eta$
Output: a BLC opening `opening` $\in \{0,1\}^{\lambda \cdot \tau \cdot (\log_2(N)+2) + \tau \cdot n \cdot \log_2 |\mathbb{F}|}$

1: $(\texttt{tree}, \texttt{ls\_com}, \Delta_x^{(1)}) \leftarrow$ Parse(`key`)
2: **for** $e = 0$ **to** $\tau - 1$ **do**
3: `path`$[e] \leftarrow$ CT.SmallGGMTree.Open(`tree`$[e], i^*[e]$)
4: `out_ls_com`$[e] \leftarrow$ `ls_com`$[e][i^*[e]]$
5: `chunk`$[e] \leftarrow$ Serialize(`path`$[e]$, `out_ls_com`$[e]$, $\Delta_x^{(1)}[e], \alpha_1[e]$)
6: `opening` $\leftarrow$ Serialize(`chunk`)
7: **return** `opening`

Algorithm 18 OT.BLC.Open(`key`, i^*, α_1)

Input: a commitment key `key`, an index array i^*, τ vectors $\alpha_1[0], \ldots, \alpha_1[\tau - 1] \in \mathbb{K}^\eta$
Output: a BLC opening `opening` $\in \{0,1\}^{3\lambda \cdot \tau + \lambda T_{\text{open}} + \tau \cdot n \cdot \log_2 |\mathbb{F}|}$

1: $(\texttt{tree}, \texttt{ls\_com}, \Delta_x) \leftarrow$ Parse(`key`)
2: `hidden_leaf_idxs` $= (i^*[0] \cdot \tau + 0, \ldots, i^*[\tau - 1] \cdot \tau + (\tau - 1))$
3: `path` $\leftarrow$ OT.BigGGMTree.Open(`tree`, `hidden_leaf_idxs`)
4: **for** $e = 0$ **to** $\tau - 1$ **do**
5: `out_ls_com`$[e] \leftarrow$ `ls_com`$[e][i^*[e]]$
6: `opening` $\leftarrow$ Serialize(`path`, `out_ls_com`, Δ_x, α_1)
7: **return** `opening`

Evaluation Computation. 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 19 CT.BLC.Eval($\texttt{salt}, \texttt{opening}, i^*$)

Input: a salt `salt`, a BLC opening `opening`, index array i^*
Output: commitment $\texttt{com}_1$, evaluation arrays `x_eval`, `u_eval`, coefficient array α_1

$\texttt{chunk} = \mathsf{Parse}(\texttt{opening}, (\{0,1\}^{|\texttt{rep\_opening}|})^\tau)$ ▷ Write $|\texttt{opening}|$ as $\tau \cdot |\texttt{rep\_opening}|$
for $e = 0$ **to** $\tau - 1$ **do**
  $(\texttt{path}[e], \texttt{out\_ls\_com}[e], \Delta_x^{(1)}[e], \alpha_1[e]) \leftarrow \mathsf{Parse}(\texttt{chunk}[e])$
  $\texttt{lseed}[e] \leftarrow \mathsf{CT.SmallGGMTree.PartiallyExpand}(\texttt{salt}, \texttt{path}[e], e, i^*[e])$
  $\Delta_x[e] = 0^\lambda \parallel \Delta_x^{(1)}[e]$
  $(\texttt{ls\_com}[e], \texttt{x\_eval}[e], \texttt{u\_eval}[e])$
    $\leftarrow \mathsf{BLC.ConvertToLineEvaluation}(\texttt{salt}, e, \texttt{lseed}[e], \Delta_x[e], i^*[e])$
  $\texttt{ls\_com}[e][i^*[e]] \leftarrow \texttt{out\_ls\_com}[e]$
  $\texttt{com}_1[e] \leftarrow \mathsf{Hash}_7(\mathsf{Bits}_8(e), \texttt{salt}, \texttt{ls\_com}[e], \Delta_x^{(1)}[e])$
return $(\texttt{com}_1, \texttt{x\_eval}, \texttt{u\_eval}, \alpha_1)$

Algorithm 20 OT.BLC.Eval($\texttt{salt}, \texttt{opening}, i^*$)

Input: a salt `salt`, a BLC opening `opening`, index array i^*
Output: commitment $\texttt{com}_1$, evaluation arrays `x_eval`, `u_eval`, coefficient array α_1

$(\texttt{path}, \texttt{out\_ls\_com}, \Delta_x, \alpha_1) = \mathsf{Parse}(\texttt{opening})$
$\texttt{hidden\_leaf\_idxs} = (i^*[0] \cdot \tau + 0, \ldots, i^*[\tau - 1] \cdot \tau + (\tau - 1))$
$\texttt{lseed\_all} \leftarrow \mathsf{OT.BigGGMTree.PartiallyExpand}(\texttt{salt}, \texttt{path}, \texttt{hidden\_leaf\_idxs})$
for $e = 0$ **to** $\tau - 1$ **do**
  for $i = 0$ **to** $N - 1$ **do**
    $\texttt{lseed}[e][i] = \texttt{lseed\_all}[\tau \cdot i + e]$
  $(\texttt{ls\_com}[e], \texttt{x\_eval}[e], \texttt{u\_eval}[e])$
    $\leftarrow \mathsf{BLC.ConvertToLineEvaluation}(\texttt{salt}, e, \texttt{lseed}[e], \Delta_x[e], i^*[e])$
  $\texttt{ls\_com}[e][i^*[e]] \leftarrow \texttt{out\_ls\_com}[e]$
  $\texttt{com}_1[e] \leftarrow \mathsf{Hash}_7(\mathsf{Bits}_8(e), \texttt{salt}, \texttt{ls\_com}[e], \Delta_x[e])$
return $(\texttt{com}_1, \texttt{x\_eval}, \texttt{u\_eval}, \alpha_1)$

2.5.4 GGM tree routines

Small GGM Tree. The GGM-tree subroutines for the correlated-tree variant are described below. The generated tree are *perfect* binary trees with N leaves, where an opening reveals *all the leaves except one*.

The routine CT.SmallGGMTree.Expand (which is called in CT.BLC.Commit) expands a GGM tree from a master 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 CT.SmallGGMTree.Open (which is called in CT.BLC.Open) extracts the sibling path from the nodes for a given hidden index. The routine CT.SmallGGMTree.PartiallyExpand (which is called in CT.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 21 CT.SmallGGMTree.Expand($\texttt{salt}, \texttt{mseed}, e, \delta$)

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

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

Algorithm 22 CT.SmallGGMTree.Open($\texttt{tree}[e], i^*[e]$)

Input: a tree node array $\texttt{tree}[e]$, a hidden leaf index $i^*[e] \in [0, N-1]$
Output: sibling path $\texttt{path}[e]$

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

Algorithm 23 CT.SmallGGMTree.PartiallyExpand($\texttt{salt}, \texttt{path}[e], e, i^*[e]$)

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

▷ Initialize nodes to $\perp$
$(\texttt{tree}[e][1], \ldots, \texttt{tree}[e][2N-1]) = (\perp, \ldots, \perp)$
▷ 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**
  $\texttt{tree}[e][i \oplus 1] \leftarrow \texttt{path}[e][j]$
  $i \leftarrow \lfloor i/2 \rfloor$
▷ Derive nodes from sibling path
for $j = 1$ **to** $\log_2(N) - 1$ **do**
  $\texttt{tweaked\_salt} = \mathsf{TweakSalt}(\texttt{salt}, 2, \mathsf{IndexIdentifier}(e, j))$
  for $k = 2^j$ **to** $2^{j+1} - 1$ **do**
    if $\texttt{tree}[e][k] \neq \perp$ **then**
      $\texttt{tree}[e][2k] \leftarrow \mathsf{SeedDerive}(\texttt{tweaked\_salt}, \texttt{tree}[e][k])$
      $\texttt{tree}[e][2k+1] \leftarrow \texttt{tree}[e][2k] \oplus \texttt{tree}[e][k]$
for $i = 0$ **to** $N - 1$ **do**
  $\texttt{lseed}[e][i] \leftarrow \texttt{tree}[e][N+i]$
return $\texttt{lseed}[e]$

Big GGM Tree. The GGM-tree subroutines for the one-tree variant are depicted hereafter. The generated trees are *complete* (but generally non-perfect) binary trees with $\tau \cdot N$ leaves and height $h_B := \lceil \log_2(\tau \cdot N) \rceil$, where the opening reveals *all the leaves except* τ.

The routine OT.BigGGMTree.Expand (which is called in OT.BLC.Commit) expands a GGM tree from a master seed with a salt. It returns the derived list of nodes and leaf seeds. The routine OT.BigGGMTree.Open (which is called in OT.BLC.Open) extracts the sibling path from the nodes for a given hidden indexes. Since the opening procedure may fail because of the rejection mechanism limiting the sibling-path size (with the parameter T_{open}), the hidden indices must first be validated using OT.BigGGMTree.IsValidOpeningSet. The routine OT.BigGGMTree.PartiallyExpand (which is called in OT.BLC.Eval) reconstructs the necessary portion of the GGM tree from a sibling path, given the salt and the hidden leaf indices.

These routines rely on auxiliary subroutines for handling leaf indices. OT.LeafPosition returns the node index of the i-th leaf, for $i \in \{0, \ldots, \tau \cdot N - 1\}$:

$$\mathsf{OT.LeafPosition}(i) = \begin{cases} 2^{h_B} + i & \text{if } i < 2\tau N - 2^h \\ \tau N + (i - (2\tau N - 2^{h_B})) & \text{otherwise.} \end{cases}$$

Similarly, the routine OT.LeafDepth returns the depth of the i-th leaf:

$$\mathsf{OT.LeafDepth}(i) = \begin{cases} h_B & \text{if } i < 2\tau N - 2^h \\ h_B - 1 & \text{otherwise.} \end{cases}$$

Finally, these routines rely on helper subroutines for manipulating sibling paths. The routine OT.GetSensitiveNodeIndexes computes the DFS-ordered list of hidden internal nodes, while

OT.GetNodeIndexesInPath returns the ordered list of node indices forming the corresponding sibling path. Let us emphasize that, unlike other schemes relying on the one-tree optimization, MQOM always reveals exactly T_{open} nodes. Consequently, *no padding mechanism* is required to fix the size of the opening.

Algorithm 24 OT.BigGGMTree.Expand(salt, mseed)

Input: a salt salt $\in \{0,1\}^\lambda$, a root seed mseed $\in \{0,1\}^\lambda$
Output: a tree node array tree, a leaf seed array lseed_all for all $\tau \cdot N$ leaves

tree[1] $\leftarrow$ mseed
for $k = 1$ **to** $(\tau \cdot N - 1)$ **do**
    tweaked_salt = TweakSalt(salt, 2, k)
    tree[$2k$] $\leftarrow$ SeedDerive(tweaked_salt, tree[k])
    tree[$2k+1$] $\leftarrow$ tree[$2k$] $\oplus$ tree[k]
for $i = 0$ **to** $(\tau \cdot N - 1)$ **do**
    lseed_all[i] $\leftarrow$ tree[OT.LeafPosition(i)]
return (tree, lseed_all)

Algorithm 25 OT.GetSensitiveNodeIndexes(hidden_leaves_idxs)

Input: indexes of the hidden leaves hidden_leaves_idxs
Output: indexes of the sensitive nodes hidden_nodes

sorted_hidden_leaves $\leftarrow$ Sort(hidden_leaves_idxs) ▷ Numerically incrementing order
hidden_nodes $\leftarrow$ List.Init() ▷ Maximal list size: $\tau \cdot (h+1)$
leaf_path[0] $= 0, \ldots,$ leaf_path[h] $= 0$
for $e = 0$ **to** $\tau - 1$ **do**
    leaf_idx $\leftarrow$ sorted_hidden_leaves[e]
    node_idx $\leftarrow$ OT.LeafPosition(leaf_idx)
    depth $\leftarrow$ OT.LeafDepth(leaf_idx)
    merge_pos = depth
    while merge_pos ≥ 0 and leaf_path[merge_pos] $!=$ node_idx **do**
        leaf_path[merge_pos] $\leftarrow$ node_idx
        node_idx $\leftarrow \lfloor$node_idx$/2 \rfloor$
        merge_pos $\leftarrow$ merge_pos $- 1$
    for $j =$ merge_pos $+ 1$ **to** depth **do**
        hidden_nodes.Append(leaf_path[j])
return hidden_nodes

Algorithm 26 OT.BigGGMTree.IsValidOpeningSet(`hidden_leaves_idxs`)

Input: indexes of the hidden leaves `hidden_leaves_idxs`
Output: Boolean indicating whether the opening set is valid

1: `hidden_nodes` $\leftarrow$ OT.GetSensitiveNodeIndexes(`hidden_leaves_idxs`) ▷ Maximal list size: $\tau \cdot (h+1)$
2: `size` $= |$`hidden_nodes`$| - 2\tau + 1$ ▷ Minimal opening size
3: **if** `size` $> T_{\text{open}}$ **then**
4: **return false**
5: **else**
6: **return true**

Algorithm 27 OT.GetNodeIndexesInPath(`hidden_nodes`)

Input: indexes of the sensitive nodes `hidden_nodes`
Assumption: `size` $\leq T_{\text{open}}$, with `size` $:= |$`hidden_nodes`$| - 2\tau + 1$
Output: indexes of the T_{open} nodes to be revealed `path_node_idxs`

1: `pos` $= 0$
2: `size` $= |$`hidden_nodes`$| - 2\tau + 1$
3: `stack` $\leftarrow$ Stack.Init() ▷ Maximal stack size: $h+1$
4: `stack`.Push(1)
5: **while** `stack`.GetLength() > 0 **do**
6: $k \leftarrow$ `stack`.Pop()
7: **if** $k \notin$ `hidden_nodes` **then**
8: **if** `size` $< T_{\text{open}}$ and IsLeaf(k) $=$ **false then** ▷ IsLeaf(k) $=$ **false** *iff* $k < \tau \cdot N$
9: `size` $\leftarrow$ `size` $+ 1$
10: **else**
11: `path_node_idxs[pos]` $= k$
12: `pos` $\leftarrow$ `pos` $+ 1$
13: **continue**
14: **if** IsLeaf(k) $=$ **false then**
15: `stack`.Push($2k+1$)
16: `stack`.Push($2k$)
17: **return** `path_node_idxs`

Algorithm 28 OT.BigGGMTree.Open(tree, hidden_leaf_idxs)

Input: a tree node array tree, hidden leaf indexes hidden_leaf_idxs
Assumption: OT.BigGGMTree.IsValidOpeningSet(hidden_leaf_idxs) = **true**
Output: sibling path path

▷ Get indexes of the sensitive nodes (at most $\tau \cdot (h+1)$ indexes)
1: hidden_nodes ← OT.GetSensitiveNodeIndexes(hidden_leaf_idxs)
▷ Get indexes of the T_{open} nodes to be revealed
2: path_node_idxs ← OT.GetNodeIndexesInPath(hidden_nodes)
▷ Collect the T_{open} nodes to be revealed
3: **for** pos $= 0$ **to** $T_{\text{open}} - 1$ **do**
4: path[pos] ← tree[path_node_idxs[pos]]
5: **return** path

Algorithm 29 OT.BigGGMTree.PartiallyExpand(salt, path, hleaves_idxs)

Input: a salt salt $\in \{0,1\}^{\lambda}$, a sibling path path, hidden leaf indexes hleaves_idxs
Assumption: OT.BigGGMTree.IsValidOpeningSet(hidden_leaf_idxs) = **true**
Output: a partial leaf seed array lseed_all

▷ Get indexes of the sensitive nodes
1: hidden_nodes ← OT.GetSensitiveNodeIndexes(hleaves_idxs) ▷ Maximal list size: $\tau \cdot (h+1)$
▷ Get indexes of the T_{open} nodes in the path
2: path_node_idxs ← OT.GetNodeIndexesInPath(hidden_nodes)
▷ Assign nodes with sibling path
3: (tree[1], ..., tree[$2\tau N - 1$]) $= (\perp, \ldots, \perp)$
4: **for** pos $= 0$ **to** $T_{\text{open}} - 1$ **do**
5: tree[path_node_idxs[pos]] ← path[pos]
▷ Derive nodes from sibling path
6: **for** $k = 1$ **to** $(\tau \cdot N - 1)$ **do**
7: **if** tree[k] $\neq \perp$ **then**
8: tweaked_salt $=$ TweakSalt(salt, 2, k)
9: tree[$2k$] ← SeedDerive(tweaked_salt, tree[k])
10: tree[$2k+1$] ← tree[$2k$] $\oplus$ tree[k]
11: **for** $i = 0$ **to** $(\tau \cdot N - 1)$ **do**
12: lseed_all[i] ← tree[OT.LeafPosition(i)]
13: **return** lseed_all

2.5.5 Seed processing routines

The following algorithms depict the subroutines of the GGM trees that are used to derive, commit and expand the seeds. By construction, the most significant bit of the tweaked salt returned by TweakSalt is always set to zero. This ensures domain separation from the grinding procedure. The SeedExpand routine has two variant-specific instantiations. The routine CT.SeedExpand expands a seed while ensuring that the first λ output bits coincide with the input seed, thereby introducing the desired correlation. In contrast, OT.SeedExpand expands the seed without introducing any correlation.

Algorithm 30 IndexIdentifier(e, j)

Input: an execution index $e \in [0, \tau - 1]$ with $\tau \leq 64$, an index $j \in [0, 2^7 - 1]$
Output: an identifier $\texttt{iip} \in [0, 2^{13} - 1]$

1: $\texttt{iip} \leftarrow e + 64 \cdot j$
2: **return** iip

Algorithm 31 TweakSalt$(\texttt{salt}, \texttt{sel}, v)$

Parameters: $|\texttt{tweak}| = 16$ with correlated trees, $|\texttt{tweak}| = 24$ with one big tree
Input: a salt $\texttt{salt} \in \{0,1\}^\lambda$, a selector $\texttt{sel} \in \{0,1,2,3\}$, an index $v \in [0, 2^{|\texttt{tweak}|-3} - 1]$
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 seed expand

1: $\texttt{tweak} \leftarrow \texttt{sel} + 4 \cdot v$
2: $\texttt{tweaked\_salt} \leftarrow \mathsf{Truncate}_{\lambda - |\texttt{tweak}|}(\texttt{salt}) \,\|\, \mathsf{Bits}_{|\texttt{tweak}|}(\texttt{tweak})$
3: **return** tweaked_salt

Algorithm 32 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 $\mathtt{new\_seed} \in \{0,1\}^\lambda$

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

Algorithm 33 SeedCommit$(\texttt{tweaked_salt}_0, \texttt{tweaked_salt}_1, \mathsf{seed})$

Input: two tweaked salts $\texttt{tweaked\_salt}_0, \texttt{tweaked\_salt}_1 \in \{0,1\}^\lambda$, a seed $\mathsf{seed} \in \{0,1\}^\lambda$
Output: a seed commitment $\texttt{seed\_com} \in \{0,1\}^{2\lambda}$

1: $\texttt{com}_1 \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt}_0, \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
2: $\texttt{com}_2 \leftarrow \mathsf{Enc}(\texttt{key} := \texttt{tweaked\_salt}_1, \texttt{ptx} := \mathsf{seed}) \oplus \psi(\mathsf{seed})$
3: $\texttt{seed\_com} \leftarrow \texttt{com}_1 \,\|\, \texttt{com}_2$
4: **return** seed_com

Algorithm 34 CT.SeedExpand($\texttt{salt}, e, \mathsf{seed}, n_{\text{bytes}}$)

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 $\mathsf{out} \in \{0,1\}^{8 \cdot n_{\text{bytes}}}$

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

Algorithm 35 OT.SeedExpand($\texttt{salt}, e, \mathsf{seed}, n_{\text{bytes}}$)

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 $\mathsf{out} \in \{0,1\}^{8 \cdot n_{\text{bytes}}}$

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

Algorithm 36 PRG($\mathsf{seed}, n_{\text{bytes}}$)

Input: a seed $\mathsf{seed} \in \{0,1\}^{\lambda}$, a number of bytes $n_{\text{bytes}} \in \mathbb{N}$
Output: a pseudorandom byte string $\texttt{out} \in \{0,1\}^{8 \cdot n_{\text{bytes}}}$

$n_{\text{blocks}} \leftarrow \lceil 8 \cdot n_{\text{bytes}} / \lambda \rceil$
for $i = 0$ **to** $n_{\text{blocks}} - 1$ **do**
    $\texttt{block}[i] \leftarrow \mathsf{Enc}(\texttt{key} := \mathsf{seed}, \texttt{ptx} := \mathsf{Bits}_{\lambda}(i))$
$(\texttt{byte}[0] \parallel \ldots \parallel \texttt{byte}[n_{\text{blocks}} \cdot (\lambda/8) - 1]) \leftarrow \mathsf{Parse}(\texttt{block}[0] \parallel \ldots \parallel \texttt{block}[n_{\text{blocks}} - 1])$
$\texttt{out} \leftarrow \texttt{byte}[0] \parallel \ldots \parallel \texttt{byte}[n_{\text{bytes}} - 1]$
return $\texttt{out}$

2.5.6 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 5 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 5: 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_eq` (MQ equations),
- Hash_1: hash commitment of α_0, α_1,
- Hash_2: hash commitment of $\mathtt{com}_1, \mathtt{com}_2$ (pre-signature identifier),
- Hash_3: message hash,
- Hash_4: Fiat-Shamir hash (signature identifier),
- XOF_5: grinding precomputation hash,
- XOF_6: opening challenge sampling,
- Hash_7: hash commitment of leaf seed commitments,
- XOF_8: generation of the batching challenge Γ.

2.5.7 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, least significant bit first. Formally, $\mathsf{Bits}_\ell(v) = (b_1, \ldots, b_\ell)$ is the unique bit-string such that

$$v = \sum_{i=1}^{\ell} 2^{i-1} \cdot b_i \ .$$

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 relies on the binary fields since its second version (MQOM v2). 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 $|\mathbb{F}| = 16$ which enjoys faster implementation. For those two fields, we took the number of equations m to be equal the number of unknowns n and selected the $m = n$ parameter to achieve a target security level (for categories I, III and V) according to the state of the art of MQ cryptanalysis (see Section 5.2).

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 simplifies the packing strategy – 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 4 for $\mathbb{F} = \mathbb{F}_{16}$.

For $|\mathbb{F}| = 16$, we additionally take into account that some values of n are more implementation-friendly. Specifically, we set n to 64, 96, and 128 for the three security levels, respectively. These choices enable efficient vectorized implementations on laptop-grade CPUs, with vector registers being fully utilized. From a structural perspective, they also ensure that the λ-bit blocks generated during seed expansion are fully utilized (see the SeedExpand routines).

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, when using correlated trees. A larger N leads to a shorter signature at the cost of slower signing and verification algorithms. We chose to consider several values for N, namely $N \in \{4096, 8192\}$ (shorter variant), $N = 2048$ (short variant) and $N = 256$ (fast variant), to obtain three 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 $\mathbb{F}_{16}$) and thus define $\mathbb{K}$ as their common extension such that $|\mathbb{K}| \geq N$. This way, we get $\mathbb{K} = \mathbb{F}_{2^{16}}$ for $N > 256$ 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 $\eta = \lambda / \log_2 |\mathbb{K}|$ to ensure a soundness error of $1/|\mathbb{K}|^\eta = 2^{-\lambda}$ for the batching. 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). Finally, the height h_L of the large tree is naturally set as $h_L = \lceil \log_2(\tau \cdot N) \rceil$, as it has $\tau \cdot N$ leaves, and we chose the smallest value for T_{open} such that the overall rejection rate remains practical.

Field extension. As explained previously, the MQOM signature scheme relies on four different fields for $\mathbb{F}$ and $\mathbb{K}$: $\mathbb{F}_2$, $\mathbb{F}_{2^4}$, $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{16}}$. Table 6 summarizes the field extensions that we use in our instances.

Table 6: Definition of field extensions.

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

At some stage, the MQOM signature requires lifting field elements into an extension field. This lifting is straightforward by construction in the cases $\mathbb{F}_2 \hookrightarrow \mathbb{F}_{2^4}$ and $\mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{16}}$, thanks to the tower field structure. However, the situation is different when lifting elements from $\mathbb{F}_{2^4}$ to $\mathbb{F}_{2^8}$. In this case, we rely on a field morphism between $\mathbb{F}_{2^4}$ and $\mathbb{F}_{2^8}$, defined such that the element $\rho \in \mathbb{F}_{2^4}$ is mapped to $\xi^7 + \xi^6 + \xi^5 \in \mathbb{F}_{2^8}$.

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, as explained in Section 2.5.3, we use the Gray-code ordering, namely

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

for all $i \in \{0, N-1\}$, where $\theta_j^{(i)} = \mathsf{i}_j \oplus \mathsf{i}_{j+1}$ for all j, with $(\mathsf{i}_0, \ldots, \mathsf{i}_{16})$ the binary decomposition of $i := \sum_{j=0}^{16} \mathsf{i}_j \cdot 2^j$.

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 $\hat{y} \in \mathbb{K}^{\hat{m}}$ corresponding to the outputs of the equations. For $|\mathbb{K}| = 256$, we store one field element on one byte. For $|\mathbb{K}| = 256^2$, we store one field element on two bytes. Thus, the size of the public key is given by:

$$|\mathtt{pk}| = \begin{cases} \frac{2\lambda}{8} + \frac{m}{8} \text{ bytes} & \text{for } \mathbb{F} := \mathbb{F}_2 \\ \frac{2\lambda}{8} + \frac{m}{2} \text{ bytes} & \text{for } \mathbb{F} := \mathbb{F}_{16} \end{cases}$$

Secret key. The secret key consists of the same elements as the public key, plus a serialized vector $x \in \mathbb{F}^n$ corresponding to the secret solution of the MQ system. For $|\mathbb{F}| = 2$, we store 8 field elements on one byte. For $|\mathbb{F}| = 16$, we store two field elements on one byte. For $|\mathbb{F}| = 256$, we store one field element on one byte. Thus, the size of the secret key is given by:

$$|\mathtt{sk}| = \begin{cases} \frac{2\lambda}{8} + \frac{m}{8} + \frac{n}{8} \text{ bytes} & \text{for } \mathbb{F}_2 \\ \frac{2\lambda}{8} + \frac{m}{2} + \frac{n}{2} \text{ bytes} & \text{for } \mathbb{F}_{16} \end{cases}$$

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 a 2λ-bit seed `seed_key` from which $(\mathtt{mseed\_eq}, x)$ would be derived through a pseudorandom generator. 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:

- when using correlated trees,

$$
\begin{aligned}
|\sigma| &= 2\lambda && \text{size of } \texttt{sig\_id} \\
&+ \lambda && \text{size of the } \texttt{salt} \\
&+ 32 && \text{size of } \texttt{nonce} \\
&+ \tau \cdot (\eta \cdot \mu \cdot \log_2 |\mathbb{F}|) && \text{size of } \alpha_1[e] \\
&+ \tau \cdot (n \cdot \log_2 |\mathbb{F}| - \lambda) && \text{size of } \Delta_x^{(1)}[e] \text{ in } \texttt{opening} \\
&+ \tau \cdot \lambda \cdot \log_2 N && \text{size of } \texttt{path}[e] \text{ in } \texttt{opening} \\
&+ \tau \cdot 2\lambda && \text{size of } \texttt{out\_ls\_com}[e] \text{ in } \texttt{opening}
\end{aligned}
$$

- when using one big tree,

$$
\begin{aligned}
|\sigma| &= 2\lambda && \text{size of } \texttt{sig\_id} \\
&+ \lambda && \text{size of the } \texttt{salt} \\
&+ 32 && \text{size of } \texttt{nonce} \\
&+ \tau \cdot (\eta \cdot \mu \cdot \log_2 |\mathbb{F}|) && \text{size of } \alpha_1[e] \\
&+ \tau \cdot (n \cdot \log_2 |\mathbb{F}|) && \text{size of } \Delta_x[e] \text{ in } \texttt{opening} \\
&+ \lambda \cdot T_{\text{open}} && \text{size of } \texttt{path} \text{ in } \texttt{opening} \\
&+ \tau \cdot 2\lambda && \text{size of } \texttt{out\_ls\_com}[e] \text{ in } \texttt{opening}
\end{aligned}
$$

Given our natural serialization of field elements, $\log_2 |\mathbb{F}|$ is 1 for $\mathbb{F}_2$ and 4 for $\mathbb{F}_{16}$. Using the fact that $\eta \cdot \mu \cdot \log_2 |\mathbb{F}| = \lambda$ by design, we obtain the following sizes in bytes:

$$|\sigma| = 4 + \frac{\tau \cdot n}{8} + \frac{\lambda}{8} \cdot \texttt{nb}_\lambda \quad \text{for } \mathbb{F}_2,$$

and

$$|\sigma| = 4 + \frac{\tau \cdot n}{2} + \frac{\lambda}{8} \cdot \texttt{nb}_\lambda \quad \text{for } \mathbb{F}_{16},$$

with $\texttt{nb}_\lambda := 3 + \tau \cdot (\log_2 N + 2)$ when using correlated trees and $\texttt{nb}_\lambda := 3 + T_{\text{open}} + 3\tau$ when using the one-tree technique.

3.3 Proposed instances

All the signature parameters are summarized in Table 7 and Table 8, while the corresponding key and signature sizes are given in Table 9.

Table 7: The MQ parameters of MQOM for NIST Security Categories I, III, and V.

Parameter Sets	**NIST Security**	**MQ Parameters**	
		$\lvert\mathbb{F}\rvert$	$m = n$
MQOM3-L1-gf2	Cat. I	2	160
MQOM3-L1-gf16	Cat. I	16	64
MQOM3-L3-gf2	Cat. III	2	240
MQOM3-L3-gf16	Cat. III	16	96
MQOM3-L5-gf2	Cat. V	2	320
MQOM3-L5-gf16	Cat. V	16	128

Table 8: The proof system parameters of MQOM for NIST Security Categories I, III, and V.

Parameter Sets	**NIST Security**	**Proof System Parameters**								
		λ	τ	N	μ	$\hat{m}$	η	w	h_B	T_{open}
MQOM3-L1-gf2-shorter-ct	Cat. I	128	10	4096	16	10	8	18	-	-
MQOM3-L1-gf2-shorter-ot	Cat. I	128	10	8192	16	10	8	8	17	110
MQOM3-L1-gf16-short-ct	Cat. I	128	12	2048	4	16	8	8	-	-
MQOM3-L1-gf16-short-ot	Cat. I	128	12	2048	4	16	8	8	15	120
MQOM3-L1-gf16-fast-ct	Cat. I	128	17	256	2	32	16	9	-	-
MQOM3-L1-gf16-fast-ot	Cat. I	128	17	256	2	32	16	9	13	119
MQOM3-L3-gf2-shorter-ct	Cat. III	192	16	4096	16	15	12	16	-	-
MQOM3-L3-gf2-shorter-ot	Cat. III	192	17	4096	16	15	12	5	17	170
MQOM3-L3-gf16-short-ct	Cat. III	192	18	2048	4	24	12	12	-	-
MQOM3-L3-gf16-short-ot	Cat. III	192	19	2048	4	24	12	2	16	175
MQOM3-L3-gf16-fast-ct	Cat. III	192	26	256	2	48	24	10	-	-
MQOM3-L3-gf16-fast-ot	Cat. III	192	26	256	2	48	24	10	13	186
MQOM3-L5-gf2-shorter-ct	Cat. V	256	22	4096	16	20	16	14	-	-
MQOM3-L5-gf2-shorter-ot	Cat. V	256	22	4096	16	20	16	14	17	241
MQOM3-L5-gf16-short-ct	Cat. V	256	25	2048	4	32	16	6	-	-
MQOM3-L5-gf16-short-ot	Cat. V	256	25	2048	4	32	16	6	16	238
MQOM3-L5-gf16-fast-ct	Cat. V	256	35	256	2	64	32	11	-	-
MQOM3-L5-gf16-fast-ot	Cat. V	256	36	256	2	64	32	4	14	235

Table 9: The key and signature sizes in bytes.

Parameter Set	**Sizes (in bytes)**		
	pk	sk	Sig.
MQOM3-L1-gf2-shorter-ct/ot	52	72	2 492 / 2 492
MQOM3-L1-gf16-short-ct/ot	64	96	2 932 / 2 932
MQOM3-L1-gf16-fast-ct/ot	64	96	3 316 / 3 316
MQOM3-L3-gf2-shorter-ct/ot	78	108	5 932 / 5 890
MQOM3-L3-gf16-short-ct/ot	96	144	6 556 / 6 556
MQOM3-L3-gf16-fast-ct/ot	96	144	7 564 / 7 660
MQOM3-L5-gf2-shorter-ct/ot	104	144	10 836 / 10 804
MQOM3-L5-gf16-short-ct/ot	128	192	12 100 / 11 716
MQOM3-L5-gf16-fast-ct/ot	128	192	13 540 / 13 380

3.4 Benchmarks

3.4.1 Benchmarks on `x86` platforms

The following `x86` platform profiles have been used for benchmarking:

- an AVX2+VAES optimized implementation, whose results are given in Table 10;
- an AVX2+VAES+GFNI optimized implementation, whose results are given in Table 11;
- an AVX-512+VAES+GFNI optimized implementation, whose results are given in Table 12.

The benchmarks were more specifically conducted on:

- Intel Core Ultra 7 265U, a laptop grade CPU which has AVX2+VAES+GFNI support;
- AMD Ryzen Threadripper PRO 7995WX, a workstation grade CPU which has AVX-512+VAES+GFNI support.

The VAES instruction set is an extension to the AES-NI one where the number of treated AES blocks exploits the full `ymm` 256-bit register on AVX2 (2 AES blocks), or the whole `zmm` 512-bit register on AVX-512 (4 AES blocks), allowing for parallel encryptions. VAES is usually present on all AMD CPUs since the Zen 3 microarchitecture (2020), and most of the Intel CPUs from Ice Lake (2019) onwards: it can be consequently considered as widespread now. The MQOM implementation takes advantage of VAES where available (and falls back to legacy AES-NI elsewhere).

The GFNI instruction set brings native $\mathbb{F}_{256}$ multiplication and inversion acceleration in the Rijndael field. This immediately benefits to the PIOP implementation with optimized field operations, as well as the BLC with AES/Rijndael key schedule improvement using field inversion. Regarding AVX-512, the MQOM implementation leverages dedicated optimizations that specifically make use of the `avx512f`, `avx512vl`, `avx512bw`, `avx512vpopcntdq` and `avx512vbmi` extensions. The cycles measurements make use of performance counters of the machine for accurate values. The timing measurements have been made with the turbo boost mode (*i.e.* automatic frequency scaling of the CPU) deactivated. All the results have been gathered using the `gcc` toolchain[1] with the `-O3` compilation option.

About GFNI and AVX-512 support across `x86` platforms: As we can see on Table 11, GFNI brings an interesting performance boost for MQOM (up to 50% on some instances). AVX-512 also allows saving cycles on compatible platforms. This raises the question of how widespread these `x86` extensions are across modern Intel and AMD processors. Although the specific support of GFNI is usually not detailed on Intel's and AMD's product briefs, we have used an open benchmarking platform [Ope] that gathers a large CPU database (more than 1700 CPUs listed from the 10 last years) with the result of the Linux command "`cat /proc/cpuinfo`" allowing to check for specific extensions.

From this database, we can see that GFNI has been introduced by Intel around Q4 2020 with the Ice Lake microarchitecture: the oldest referenced CPU supporting it is the Intel Core i5-1038NG7. Since Q1 2022, most of Intel CPUs seem to support AVX2 with GFNI, from Core i3 laptop to Xeon server grade ones (even the low-end Intel Celeron N5095 embeds it). AVX-512

[1] In version `16.2.0` for the Intel Core Ultra 7 265U machine, and in version `15.3.0` for the AMD Ryzen Threadripper PRO 7995WX machine.

support (with its various sub-extensions) is more erratic, as some low-end processors such as Core i3-1115G4 from Q4 2020 seem to support them, but more recent high-end CPUs such as the Meteor Lake Intel Core Ultra 7 165U or the Arrow Lake Intel Core Ultra 7 265U (that we use for our benchmarks) do not support them anymore. This seems to be related to the fact that AVX-512 was not fuse-disabled on early consumer grade Alder Lake CPUs, leading to a bypass activating them in the BIOS [Wik]. In fact, Intel officially reserves AVX-512 support exclusively for its Xeon server-grade processors.

Regarding AMD, the situation appears more straightforward with GFNI and AVX-512 usually coming together. They have been introduced only on server grade CPUs in the Zen 4 microarchitecture in 2023 (this is the case for our AMD Ryzen Threadripper PRO 7995WX benchmarking CPU). With the introduction of Zen 5 in Q2 2024, all the AMD CPUs from laptop grade to server grade support both GFNI and AVX-512. The only exceptions may be certain custom SoCs that AMD manufactures for specific customers, where these units are disabled to reduce silicon area or power consumption; however, this applies only to a marginal set of devices.

All in all, homogeneous GFNI support across `x86` has become a reality in 2025, which MQOM can take advantage of. AVX-512 has also been somewhat democratized with the advent of AMD Zen 5, although Intel continues to reserve it for its server-grade processors.

Table 10: Benchmark of the **optimized implementation** of MQOM on an **AVX2+VAES** machine. Timings were collected on an Intel Core Ultra 7 265U (with compilation for the AVX2 restricted instructions set using the `'-maes -mvaes -mavx2'` compilation flags).

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM3-L1-gf2-shorter-ct	0.08	0.21M	8.42	17.5M	4.61	9.72M
MQOM3-L1-gf2-shorter-ot	0.09	0.20M	14.70	30.8M	14.61	30.5M
MQOM3-L1-gf16-short-ct	0.03	0.09M	2.23	4.63M	2.15	4.51M
MQOM3-L1-gf16-short-ot	0.06	0.09M	4.10	8.57M	4.10	8.60M
MQOM3-L1-gf16-fast-ct	0.04	0.11M	0.89	1.83M	0.83	1.73M
MQOM3-L1-gf16-fast-ot	0.05	0.12M	1.24	2.54M	1.17	2.38M
MQOM3-L3-gf2-shorter-ct	0.70	1.58M	24.12	50.4M	21.06	44.1M
MQOM3-L3-gf2-shorter-ot	0.79	1.58M	40.63	84.9M	38.32	79.8M
MQOM3-L3-gf16-short-ct	0.23	0.49M	10.93	22.9M	10.56	22.1M
MQOM3-L3-gf16-short-ot	0.24	0.49M	20.50	42.9M	20.03	41.8M
MQOM3-L3-gf16-fast-ct	0.24	0.54M	3.86	8.09M	3.67	7.65M
MQOM3-L3-gf16-fast-ot	0.26	0.54M	5.28	11.0M	5.03	10.5M
MQOM3-L5-gf2-shorter-ct	1.36	2.71M	43.78	91.5M	41.27	86.5M
MQOM3-L5-gf2-shorter-ot	1.21	2.71M	64.86	136M	62.44	130M
MQOM3-L5-gf16-short-ct	0.43	0.85M	18.50	38.7M	17.85	37.4M
MQOM3-L5-gf16-short-ot	0.40	0.85M	30.31	63.5M	29.65	61.7M
MQOM3-L5-gf16-fast-ct	0.46	0.92M	8.63	17.3M	7.98	16.2M
MQOM3-L5-gf16-fast-ot	0.47	0.94M	11.20	22.8M	9.89	20.0M

Table 11: Benchmark of the **optimized implementation** of MQOM on an **AVX2+VAES+GFNI** machine. Timings were collected on an Intel Core Ultra 7 265U with the `'-march=native -mtune=native'` compilation flags.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM3-L1-gf2-shorter-ct	0.10	0.21M	7.26	15.3M	3.64	7.47M
MQOM3-L1-gf2-shorter-ot	0.14	0.21M	9.08	19.1M	8.89	18.6M
MQOM3-L1-gf16-short-ct	0.01	0.07M	1.75	3.76M	1.84	3.70M
MQOM3-L1-gf16-short-ot	0.04	0.07M	2.28	4.79M	2.34	4.85M
MQOM3-L1-gf16-fast-ct	0.04	0.09M	0.56	1.15M	0.52	1.12M
MQOM3-L1-gf16-fast-ot	0.06	0.09M	0.65	1.39M	0.61	1.26M
MQOM3-L3-gf2-shorter-ct	0.80	1.59M	18.54	38.8M	15.57	32.6M
MQOM3-L3-gf2-shorter-ot	0.75	1.59M	23.34	48.6M	20.95	43.9M
MQOM3-L3-gf16-short-ct	0.21	0.43M	9.14	19.0M	8.64	18.2M
MQOM3-L3-gf16-short-ot	0.22	0.44M	12.11	25.3M	11.86	24.7M
MQOM3-L3-gf16-fast-ct	0.26	0.48M	2.40	5.07M	2.32	4.88M
MQOM3-L3-gf16-fast-ot	0.26	0.47M	2.76	5.79M	2.68	5.53M
MQOM3-L5-gf2-shorter-ct	1.31	2.74M	27.34	57.3M	25.40	53.1M
MQOM3-L5-gf2-shorter-ot	1.31	2.73M	33.61	70.1M	31.52	65.9M
MQOM3-L5-gf16-short-ct	0.34	0.74M	12.72	26.6M	12.24	25.6M
MQOM3-L5-gf16-short-ot	0.38	0.73M	15.93	33.2M	15.41	32.3M
MQOM3-L5-gf16-fast-ct	0.35	0.78M	4.53	8.75M	4.24	8.32M
MQOM3-L5-gf16-fast-ot	0.33	0.78M	5.79	11.3M	4.80	9.33M

Table 12: Benchmark of the **optimized implementation** of MQOM on an **AVX-512+VAES+GFNI** machine. Timings were collected on an AMD Ryzen Threadripper PRO 7995WX with the `'-march=native -mtune=native'` compilation flags.

Instance	KeyGen		Sign		Verify	
	ms	cycles	ms	cycles	ms	cycles
MQOM3-L1-gf2-shorter-ct	0.05	0.13M	5.22	12.8M	2.39	5.97M
MQOM3-L1-gf2-shorter-ot	0.04	0.13M	6.24	15.4M	6.09	14.9M
MQOM3-L1-gf16-short-ct	0.03	0.05M	1.15	2.81M	1.10	2.78M
MQOM3-L1-gf16-short-ot	0.04	0.05M	1.53	3.76M	1.52	3.82M
MQOM3-L1-gf16-fast-ct	0.04	0.09M	0.43	0.96M	0.33	0.93M
MQOM3-L1-gf16-fast-ot	0.06	0.10M	0.49	1.19M	0.43	1.13M
MQOM3-L3-gf2-shorter-ct	0.69	1.53M	14.01	34.7M	12.07	29.9M
MQOM3-L3-gf2-shorter-ot	0.57	1.53M	16.96	41.8M	15.21	37.5M
MQOM3-L3-gf16-short-ct	0.13	0.42M	7.24	17.8M	6.93	17.1M
MQOM3-L3-gf16-short-ot	0.19	0.42M	8.94	22.3M	8.80	21.5M
MQOM3-L3-gf16-fast-ct	0.16	0.47M	1.91	4.79M	1.94	4.63M
MQOM3-L3-gf16-fast-ot	0.18	0.47M	2.12	5.23M	2.07	5.09M
MQOM3-L5-gf2-shorter-ct	1.09	2.48M	18.32	45.4M	17.08	42.2M
MQOM3-L5-gf2-shorter-ot	0.99	2.48M	21.79	53.7M	20.62	50.9M
MQOM3-L5-gf16-short-ct	0.26	0.66M	8.78	21.8M	8.60	21.2M
MQOM3-L5-gf16-short-ot	0.28	0.66M	11.26	27.7M	10.47	25.9M
MQOM3-L5-gf16-fast-ct	0.31	0.73M	3.73	7.62M	3.31	7.18M
MQOM3-L5-gf16-fast-ot	0.28	0.74M	4.66	9.76M	3.62	7.93M

3.4.2 Benchmarks on ARM AArch64 platforms

We provide an AArch64 optimized implementation of MQOM (more specifically for NEON armv8-a+crypto+sha3), that uses the following accelerations:

- the NEON SIMD instruction set, mostly used for the fields implementation (taking advantage of carryless multiplications);
- the AES instruction set armv8-a+crypto;
- the SHA3 instruction set armv8-a+sha3.

We have benchmarked this implementation of MQOM on an Apple Silicon M2 at 3.4 GHz with 16GB of RAM running under macOS 15.7.5: the results are presented on Table 13. The performance results are only in milliseconds (ms) since cycles measurements are not easy to collect without administrative rights. The compiler is Apple clang version 17.0.0, and the options are `-O3 -march=armv8-a+crypto+sha3`.

Table 13: Benchmark of the **AArch64 optimized implementation** of the MQOM on an Apple M2 machine (timings in milliseconds). Timings were collected with the `'-march=armv8-a+crypto+sha3'` compilation flags.

Instance	KeyGen ms	Sign ms	Verify ms
MQOM3-L1-gf2-shorter-ct	0.05	3.67	2.69
MQOM3-L1-gf2-shorter-ot	0.05	6.91	6.24
MQOM3-L1-gf16-short-ct	0.02	1.36	1.28
MQOM3-L1-gf16-short-ot	0.02	1.79	1.60
MQOM3-L1-gf16-fast-ct	0.03	0.41	0.38
MQOM3-L1-gf16-fast-ot	0.03	0.49	0.44
MQOM3-L3-gf2-shorter-ct	0.50	12.73	11.06
MQOM3-L3-gf2-shorter-ot	0.51	17.38	15.80
MQOM3-L3-gf16-short-ct	0.13	6.49	5.69
MQOM3-L3-gf16-short-ot	0.13	8.84	8.07
MQOM3-L3-gf16-fast-ct	0.15	1.80	1.64
MQOM3-L3-gf16-fast-ot	0.15	2.06	1.90
MQOM3-L5-gf2-shorter-ct	0.88	22.04	20.60
MQOM3-L5-gf2-shorter-ot	0.89	26.72	24.89
MQOM3-L5-gf16-short-ct	0.22	10.10	9.45
MQOM3-L5-gf16-short-ot	0.22	12.56	11.93
MQOM3-L5-gf16-fast-ct	0.25	3.42	3.25
MQOM3-L5-gf16-fast-ot	0.25	4.17	3.61

3.4.3 Benchmarks on embedded platforms

Benchmarking platform and methodology: We present hereafter some benchmarks on Cortex-M4 MCUs. We use a Nucleo-L4R5ZI board featuring a STM32L4R5ZI MCU with embedded 2MB of flash and 640KB of SRAM, as well as a custom board based on a STM32F437 MCU with 2MB of flash and 256KB of SRAM. All the results have been obtained with the `arm-none-eabi-gcc` toolchain in version `14.2.1` using the compilation flags `-O3` and `-march=armv7e-m -mtune=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16`.

Our benchmarking setup closely follows that of the PQM4 framework [KPR+]: we configure the MCUs at frequencies with a zero wait-state flash to remove effects of the prefetchers and have proper cycles measurements. Beyond cycle counts, we also measure SRAM usage through:

- stack usage: measured via stack spraying, and corresponding to the most consuming usage of a calling sequence;
- global variable usage: measured at link time using the compiling toolchain, by dedicating a specific ELF section of the firmware to these variables;
- dynamic allocation usage: measured by tracking `malloc` and `free` calls through dedicated wrappers.

The memory consumption reported in the following benchmarks corresponds to the sum of these three components. This figure typically represents an overestimation, since the program

does not necessarily use all global variables or allocated variables along its most stack-consuming call path. Note that this memory consumption does not include signature or key-pair sizes, as our goal is solely to assess the internal memory usage of the primitives.

For our embedded benchmarks, we focus on the Fast and Short instances of MQOM, which provide the most relevant performance trade-offs for constrained devices. The Shorter instances are nevertheless fully supported by our implementation.

Speed and memory trade-offs: The MQOM implementation has been adapted[2] to fit embedded platforms with low-memory profiles, using dedicated memory optimizations for the PRG, PIOP and BLC primitives. Some elements are described in Section 4, but for all the details of these optimizations please refer to [BF26]: although this article concerns MQOM version 2.1, all the techniques described there can be applied to MQOM version 3 to have various trade-offs between speed and memory consumption, by selecting various possible implementation strategies for the basic blocks of the algorithm. Here is an overview of these implementation choices (please check [BF26] for the complete list and rationale):

- For the AES/Rijndael symmetric primitive, we have the choice between:
  - A table-based implementation that is only constant time on platforms where table lookups are constant time (e.g. Cortex-M4 without a cache and tables in SRAM). Specifically for the L1 security level, optimized ARM Cortex-M assembly is used [SS16]. Since all the key schedules in MQOM only deal with public data, non constant-time key schedule can always be used.
  - A constant-time bitslice implementation, adapted from [Por]. Specifically for the L1 security level, optimized ARM Cortex-M assembly is used [AP20] (so-called "fixsliced" AES).
  - Specifically for the L1 security level and the STM32F437 MCU, a hardware accelerated implementation for AES-128, taking advantage of the `CRYP` engine in DMA (Direct Memory Access) mode to benefit from large buffers encryptions under the same key.
- For PIOP and the field matrix multiplication implementation in PIOP, we use a streaming generation of the matrices. The τ repetitions are processed in configurable batches, the equation matrix being streamed once per batch and shared by all its repetitions: memory scales with the batch size while the matrix is regenerated $\lceil \tau / batch \rceil$ times. Signing and verification are tuned independently. Also, we choose between:
  - A table-based implementation of $\mathbb{F}_{256}$ multiplication, either based on small log/exp lookup tables, or on the large 65kB table.
  - A SWAR (SIMD Within a Register) implementation of this multiplication (4-ways using the 32-bit registers).
  - A bitslice implementation that takes advantage of the τ repetitions in PIOP for a one-shot matrix multiplication computation, by packing the τ values in 32 bits registers and use composite representations of the Rijndael field as well as fast table-lookups on public data.
- For BLC and the GGM trees, DFS (Depth Fist Search) is used to allow for a streamed traversal of the leafs. However, many trade-offs can be leveraged:

[2] https://github.com/mqom/mqom-embedded/

- The nodes computation can be batched across the tree levels (at the expense of memory), and take advantage of the underlying symmetric encryption primitive (e.g. bitslice AES is optimal with an even number of blocks encryption, etc.). The key schedules for the nodes derivation can also be optionally cached.
- In the Correlated trees setting, DFS is composed with nodes computations under the same key (at the same level) in low tree levels to take advantage of fast ECB encryption primitives.
- In the One big tree setting, executions can be processed in batches: the tree is fully traversed $\lceil \tau / batch \rceil$ times, out-of-batch leaves being derived and discarded (and memory scales with the batch size in exchange).
- The leaf seeds can be batched before being committed and expanded in parallel (at the expense of memory). Also, the scheduled keys for the leafs commitment and expansion can be optionally cached.
- The folding can be performed in a fast fashion (accumulate first then multiply, with a buffer for accumulation) or a slow fashion (accumulate and multiply without storing a large buffer).

Among all the possible trade-offs, four representative profiles are exhibited here:

- **LUT.** This profile allows the use of look-up tables for sensitive data access in SRAM. It uses bitslicing for matrix multiplication, as it is always better in speed and size than LUT based $\mathbb{F}_{256}$ multiplication, and table-based block cipher. BLC memory trade-off is moderate ($2^5 = 32$ leaves batched, adjusted caching depending on the tree variant).
- **Balanced.** This profile seeks an interesting balanced trade-off between memory and speed. It will use bitslicing for multiplication and fixsliced block cipher. BLC memory trade-off is moderate ($2^5 = 32$ leaves batched for the Fast variant, $2^6 = 64$ for Short, adjusted caching depending on the tree variant).
- **Memory.** This profile seeks to minimize the memory footprint, at the cost of slower algorithms. It will use SWAR for matrix multiplication and fixsliced block cipher. All the existing code options are set such as the memory footprint is minimized (*e.g.* minimizing the number of batches for PIOP and BLC, removing the key schedule caches, etc.), while avoiding significant slow-downs. Specifically for BLC, only batch 2 leaves.
- **Hardware.** This profile has access to an AES-128 hardware accelerator to speed up the computation of the block cipher, hence this only concerns L1. For PIOP and BLC, we use the same parameters as the Balanced profile.

The rationale of providing these profiles is to have a choice between implementations depending on the usage context: constant-time or not, speed over memory (or the opposite), hardware accelerated when possible. More specifically for constant-time: the property is ensured whatever the underlying Cortex-M4 platform is. Using lookup tables in SRAM is constant time when there is no prefetching or caching technology between the ARM core and the peripherals (SRAM in this case). In the STM32 Cortex-M4 MCU line, this is usually the case and the two first implementation variants should be safe.[3] However, in the general case for Cortex-M4

[3]This is not the case for the flash peripheral whose access latency depends on the MCU configuration, and for which prefetchers are used to accelerate the accesses.

hardware instantiations, a constant-time access to SRAM is not guaranteed. Hence, among all the profiles, only the LUT one is not constant-time (at least it is dependent on the underlying platform).

The results are provided on Table 14 for the LUT profile, on Table 15 for the Balanced profile, on Table 16 for the Memory profile, and on Table 17 for the Hardware profile. Specifically for One big Tree implementations, we explore two BLC batching: 1 where the τ repetitions are performed in one batch, 2 where these repetitions are performed in two batches. It is to be noted that a fixed number of batches (chosen here) allows to moderate the cycles consumption, while a fixed batch size allows to keep a fixed memory budget for BLC. It is noticeable that all the benchmarked Memory profile implementations of MQOM variants of One big Tree for the security category L1, and all the variants of Correlated Trees for all the security categories, fit in **less than 14kB** of SRAM.

Table 14: Benchmark of the **LUT** profile of MQOM on a Cortex-M4, at the board's zero-wait-state benchmark clock. Memory is the peak stack plus global tables usage; the Batching column is the number of traversals of the One big Tree.

Instance	Batching	KeyGen		Sign		Verify	
		kB	cycles	kB	cycles	kB	cycles
MQOM3-L1-gf16-short-ct	-	3.38⋆	7.47M⋆	16.47⋆	213.28M⋆	16.24	210.41M
MQOM3-L1-gf16-short-ot	1	3.38⋆	7.47M⋆	26.42⋆	232.82M⋆	25.82	224.82M
MQOM3-L1-gf16-short-ot	2	3.38⋆	7.47M⋆	17.50⋆	269.69M⋆	17.64	257.19M
MQOM3-L1-gf16-fast-ct	-	3.24⋆	9.10M⋆	12.18⋆	56.81M⋆	10.91	54.37M
MQOM3-L1-gf16-fast-ot	1	3.24⋆	9.10M⋆	30.89⋆	61.07M⋆	30.34	56.84M
MQOM3-L1-gf16-fast-ot	2	3.24⋆	9.10M⋆	20.30⋆	67.60M⋆	20.58	62.57M
MQOM3-L3-gf16-short-ct	-	7.94⋆	52.20M⋆	30.23⋆	1255.23M⋆	27.74	1228.04M
MQOM3-L3-gf16-short-ot	1	7.94⋆	52.20M⋆	54.10⋆	1392.09M⋆	54.78	1360.80M
MQOM3-L3-gf16-short-ot	2	7.94⋆	52.20M⋆	38.62⋆	1633.13M⋆	38.75	1587.79M
MQOM3-L3-gf16-fast-ct	-	7.74⋆	57.94M⋆	24.31⋆	311.07M⋆	21.40	300.12M
MQOM3-L3-gf16-fast-ot	1	7.74⋆	57.94M⋆	61.74⋆	328.44M⋆	62.44	310.65M
MQOM3-L3-gf16-fast-ot	2	7.74⋆	57.94M⋆	41.21⋆	369.60M⋆	41.42	348.78M
MQOM3-L5-gf16-short-ct	-	7.58⋆	97.14M⋆	40.14⋆	1891.86M⋆	36.84	1876.18M
MQOM3-L5-gf16-short-ot	1	7.58⋆	97.14M⋆	88.31⋆	2012.96M⋆	87.44	1969.20M
MQOM3-L5-gf16-short-ot	2	7.58⋆	97.14M⋆	57.20⋆	2317.82M⋆	57.47	2260.26M
MQOM3-L5-gf16-fast-ct	-	7.32⋆	107.95M⋆	32.89⋆	544.07M⋆	29.04	515.78M
MQOM3-L5-gf16-fast-ot	1	7.32⋆	107.95M⋆	105.08⋆	588.64M⋆	105.01	550.71M
MQOM3-L5-gf16-fast-ot	2	7.32⋆	107.96M⋆	64.50⋆	643.72M⋆	65.19	602.24M

⋆ Performance results rely on LUT use over secret entries.

Table 15: Benchmark of the **Balanced** profile of MQOM on a Cortex-M4, at the board's zero-wait-state benchmark clock. Memory is the peak stack plus global tables usage; the Batching column is the number of traversals of the One big Tree.

Instance	Batching	KeyGen		Sign		Verify	
		kB	cycles	kB	cycles	kB	cycles
MQOM3-L1-gf16-short-ct	-	2.37	6.42M	16.35	327.39M	15.28	210.02M
MQOM3-L1-gf16-short-ot	1	2.37	6.42M	26.50	377.50M	24.78	224.51M
MQOM3-L1-gf16-short-ot	2	2.37	6.42M	17.68	457.41M	16.51	256.92M
MQOM3-L1-gf16-fast-ct	-	2.22	7.75M	11.15	79.22M	9.89	54.16M
MQOM3-L1-gf16-fast-ot	1	2.22	7.75M	30.74	87.07M	29.32	56.61M
MQOM3-L1-gf16-fast-ot	2	2.22	7.75M	20.44	101.26M	19.49	62.29M
MQOM3-L3-gf16-short-ct	-	6.93	48.68M	31.28	2267.16M	28.31	1227.62M
MQOM3-L3-gf16-short-ot	1	6.93	48.68M	54.89	2643.55M	54.09	1358.94M
MQOM3-L3-gf16-short-ot	2	6.93	48.68M	40.26	3221.83M	37.66	1585.38M
MQOM3-L3-gf16-fast-ct	-	6.71	53.47M	24.30	505.82M	20.32	299.73M
MQOM3-L3-gf16-fast-ot	1	6.71	53.47M	61.81	545.71M	61.42	309.86M
MQOM3-L3-gf16-fast-ot	2	6.71	53.47M	42.79	644.65M	40.42	347.88M
MQOM3-L5-gf16-short-ct	-	6.58	88.51M	41.58	3297.88M	38.34	1874.01M
MQOM3-L5-gf16-short-ot	1	6.58	88.51M	89.42	3663.33M	86.41	1966.63M
MQOM3-L5-gf16-short-ot	2	6.58	88.51M	59.02	4415.80M	56.38	2258.00M
MQOM3-L5-gf16-fast-ct	-	6.30	96.56M	32.35	818.23M	28.46	526.09M
MQOM3-L5-gf16-fast-ot	1	6.30	96.56M	106.03	905.81M	103.98	558.69M
MQOM3-L5-gf16-fast-ot	2	6.30	96.56M	66.26	1041.23M	64.18	610.24M

Table 16: Benchmark of the **Memory** profile of MQOM on a Cortex-M4, at the board's zero-wait-state benchmark clock. Memory is the peak stack plus global tables usage; the Batching column is the number of traversals of the One big Tree.

Instance	Batching	KeyGen		Sign		Verify	
		kB	cycles	kB	cycles	kB	cycles
MQOM3-L1-gf16-short-ct	-	1.34	6.41M	6.52	406.10M	4.61	232.33M
MQOM3-L1-gf16-short-ot	1	1.34	6.42M	11.54	692.21M	11.20	339.65M
MQOM3-L1-gf16-short-ot	2	1.34	6.42M	8.16	841.78M	7.43	377.12M
MQOM3-L1-gf16-fast-ct	-	1.19	7.74M	6.40	116.88M	4.42	90.24M
MQOM3-L1-gf16-fast-ot	1	1.19	7.74M	13.79	146.87M	13.65	102.19M
MQOM3-L1-gf16-fast-ot	2	1.19	7.74M	9.81	198.16M	9.00	108.70M
MQOM3-L3-gf16-short-ct	-	2.58	49.40M	11.52	2737.81M	8.33	1542.33M
MQOM3-L3-gf16-short-ot	1	2.58	49.40M	21.78	3955.10M	21.72	1946.00M
MQOM3-L3-gf16-short-ot	2	2.58	49.40M	17.07	5023.80M	14.87	2224.63M
MQOM3-L3-gf16-fast-ct	-	2.36	54.21M	11.24	858.04M	7.58	678.52M
MQOM3-L3-gf16-fast-ot	1	2.36	54.20M	25.28	817.05M	25.14	721.95M
MQOM3-L3-gf16-fast-ot	2	2.36	54.21M	18.88	1265.37M	16.38	810.50M
MQOM3-L5-gf16-short-ct	-	2.22	89.76M	13.56	4393.15M	8.78	2751.51M
MQOM3-L5-gf16-short-ot	1	2.22	89.76M	33.50	5624.81M	33.20	3248.46M
MQOM3-L5-gf16-short-ot	2	2.22	89.75M	23.58	7210.23M	20.61	3542.48M
MQOM3-L5-gf16-fast-ct	-	1.94	97.86M	13.81	1615.29M	8.48	1469.72M
MQOM3-L5-gf16-fast-ot	1	1.94	97.87M	40.50	1388.94M	41.04	1540.01M
MQOM3-L5-gf16-fast-ot	2	1.94	97.87M	27.58	2133.13M	24.12	1695.58M

Table 17: Benchmark of the **Hardware** profile of MQOM on a Cortex-M4, at the board's zero-wait-state benchmark clock. Memory is the peak stack plus global tables usage; the Batching column is the number of traversals of the One big Tree.

Instance	Batching	KeyGen		Sign		Verify	
		kB	cycles	kB	cycles	kB	cycles
MQOM3-L1-gf16-short-ct	-	1.14†	5.01M†	14.76†	161.29M†	13.99†	158.47M†
MQOM3-L1-gf16-short-ot	1	1.14†	5.01M†	23.65†	170.35M†	23.33†	163.98M†
MQOM3-L1-gf16-short-ot	2	1.14†	5.01M†	14.98†	197.85M†	15.06†	187.71M†
MQOM3-L1-gf16-fast-ct	-	0.99†	6.27M†	9.70†	46.85M†	8.82†	44.76M†
MQOM3-L1-gf16-fast-ot	1	0.99†	6.27M†	27.93†	48.85M†	27.94†	45.56M†
MQOM3-L1-gf16-fast-ot	2	0.99†	6.27M†	17.43†	53.21M†	18.10†	49.29M†

† Uses hardware acceleration for AES-128 (only on the LEIA board).

4 Implementation aspects

In this section, we discuss several implementation aspects of MQOM, including optimization techniques to speed up computations, reduce memory usage, improve latency, and more generally enhance implementation efficiency.

4.1 Optimized folding using Gray codes

While $u_0[e]$, $u_1[e]$, $x_0[e]$ and $\Delta_x[e]$ in CT.BLC.Commit can be computed using generic field operations over $\mathbb{K}$, the *order* of the public evaluation points $\omega_0, \ldots, \omega_{N-1}$ from Ω has been chosen to enable optimization. Using the classical lexicographic order yields:

- computational complexity $O(N \cdot \log_2 N)$ with a memory complexity $O(1)$ when using direct computation;
- computational complexity $O(N + \log_2 N)$ with a memory complexity $O(N)$ when using the divide-and-conquer method [Roy22].

Although memory scaling in $O(N)$ is acceptable on laptops or servers, it can be problematic for embedded devices with limited memory. In MQOM, as in the round-2 SDitH candidate [ABB+24], we use a Gray code ordering that achieves the best of both worlds with a computational complexity of $O(N + \log_2 N)$ and memory usage of $O(1)$. By definition, the Gray code ordering ensures that two consecutive values differ in exactly one bit (see Section 3.1 for the precise definition of $\omega_1, \ldots, \omega_{N-1}$).

Let us fix a loop iteration $e \in \{0, \ldots, \tau - 1\}$, we define:

$$\mathtt{raw}_i := \mathsf{SeedExpand}(\mathtt{salt}, e, \mathtt{lseed}[i], |x|_8 + |u|_8)$$

so that $(\bar{x}_i, \bar{u}_i) = \mathsf{Parse}(\mathtt{raw}_i)$. The procedure BLC.ComputeFolding, depicted below, takes as input the raw random tapes $\mathtt{raw}_0, \ldots, \mathtt{raw}_{N-1}$ and efficiently computes the following values:

$$\bar{x}_{\text{ACC}} := \sum_i \bar{x}_i, \quad \bar{u}_{\text{ACC}} := \sum_i \bar{u}_i, \quad \bar{x}_{\text{FOLD}} := \sum_i \omega_i \cdot \bar{x}_i, \quad \bar{u}_{\text{FOLD}} := \sum_i \omega_i \cdot \bar{u}_i \ .$$

This procedure is invoked in the routine BLC.ConvertToLine (within the signing algorithm) and in the routine BLC.ConvertToLineEvaluation (within the verification algorithm). In the latter case, the random tape of the hidden leaf $i^*[e]$ is unknown, so the procedure is called with $\mathtt{raw}_{i^*[e]} := 0$. This amounts to computing the above values as sums over all indices $i \neq i^*[e]$.

In CT.BLC.Commit (signing algorithm), these values directly yield the line coefficients and correction value as:

$$u_0[e] = -\bar{u}_{\text{FOLD}} \ , \quad u_1[e] = \bar{u}_{\text{ACC}} \ , \quad x_0[e] = -\bar{x}_{\text{FOLD}} \ , \quad \Delta_x[e] = x - \bar{x}_{\text{ACC}} \ .$$

In CT.BLC.Eval (verification algorithm), these values are used to compute the evaluations v_x and v_u as:

$$\begin{aligned} v_x &= (\Delta_x[e] + \bar{x}_{\text{ACC}}) \cdot r - \bar{x}_{\text{FOLD}} \\ v_u &= \bar{u}_{\text{ACC}} \cdot r - \bar{u}_{\text{FOLD}} \ . \end{aligned}$$

From these equations, one can observe that the contributions of $\bar{x}_{i^*[e]}$ (resp. $\bar{u}_{i^*[e]}$) cancel out in the computation of v_x (resp. v_u). Hence, setting $\mathtt{raw}_{i^*[e]} = 0$ does not affect the final result.

Description of the optimized folding. The procedure BLC.ComputeFolding works as follows. After i iterations of the main loop:

- the variable $\texttt{acc}$ stores the partial sum $\sum_{k=0}^{i} \texttt{raw}_k$;
- the variables $\{\texttt{fd}_0, \ldots, \texttt{fd}_{\log_2 N-1}\}$ represent the value

$$\sum_{j=0}^{i}(w_j - w_{j+1}) \cdot \sum_{k=0}^{i} \texttt{raw}_k = \sum_{k=0}^{i} w_k \cdot \texttt{raw}_k - w_{i+1} \cdot \sum_{k=0}^{i} \texttt{raw}_k \ ,$$

  where $\texttt{fd}_j$ corresponds to the j^{th} coordinate of this value in the canonical $\mathbb{F}_2$-basis of $\mathbb{K}$.

At the end of the loop, we have $\texttt{acc} = \sum_{k=0}^{N-1} \texttt{raw}_k$ and $\{\texttt{fd}_j\}_j$ encoding $\sum_{k=0}^{N-1} w_k \cdot \texttt{raw}_k$, with the convention $w_N := 0$. The final desired values are then obtained through parsing and scaling.

Algorithm 37 BLC.ComputeFolding$(\texttt{raw}_0, \ldots, \texttt{raw}_{N-1})$

Input: $\texttt{raw}_0, \ldots, \texttt{raw}_{N-1} \in \{0,1\}^{n+\mu\cdot\eta}$

1: $\texttt{acc} = 0$ ▷ $\texttt{acc} \in \{0,1\}^{n+\mu\cdot\eta}$
2: $\texttt{fd}_0 = 0, \ldots, \texttt{fd}_{\log_2 N-1} = 0$ ▷ $\texttt{fd}_j \in \{0,1\}^{n+\mu\cdot\eta}$
3: **for** $i = 0$ **to** $N-1$ **do**
4: $\quad \texttt{acc} \leftarrow \texttt{acc} \oplus \texttt{raw}_i$
5: $\quad \texttt{fd}_{p_i} \leftarrow \texttt{fd}_{p_i} \oplus \texttt{acc}$ ▷ p_i is the position of the only bit set in $\omega_i \oplus \omega_{i+1}$
6: $(\bar{x}_{\text{ACC}}, \bar{u}_{\text{ACC}}) \leftarrow$ Parse$(\texttt{acc})$ ▷ $\bar{x}_{\text{ACC}} \in \mathbb{F}^n$, $\bar{u}_{\text{ACC}} \in \mathbb{K}^\eta$
7: $\bar{x}_{\text{FOLD}} = 0, \bar{u}_{\text{FOLD}} = 0$ ▷ $\bar{x}_{\text{FOLD}} \in \mathbb{K}^n$, $\bar{u}_{\text{FOLD}} \in \mathbb{K}^\eta$
8: **for** $j = 0$ **to** $\log_2 N - 1$ **do**
9: $\quad (\texttt{fd}_j^x, \texttt{fd}_j^u) \leftarrow$ Parse$(\texttt{fd}_j)$ ▷ $\texttt{fd}_j^x \in \mathbb{F}^n$, $\texttt{fd}_j^u \in \mathbb{K}^\eta$
10: $\quad \bar{x}_{\text{FOLD}} \leftarrow \bar{x}_{\text{FOLD}} + e_j \cdot \texttt{fd}_j^x$ ▷ $\{e_0, \ldots, e_{\log_2 |\mathbb{K}|-1}\}$ is the canonical $\mathbb{F}_2$-basis of $\mathbb{K}$
11: $\quad \bar{u}_{\text{FOLD}} \leftarrow \bar{u}_{\text{FOLD}} + e_j \cdot \texttt{fd}_j^u$
12: **return** $(\bar{x}_{\text{ACC}}, \bar{u}_{\text{ACC}}), (\bar{x}_{\text{FOLD}}, \bar{u}_{\text{FOLD}})$

Incremental description of BLC.ConvertToLine. On memory-constrained devices, the routine BLC.ComputeFolding should be implemented incrementally in order to avoid a large memory footprint. Algorithm 38 presents an alternative description of BLC.ConvertToLine in which the folding computation is performed on the fly. More precisely, after each leaf seed is expanded, the accumulators $\texttt{acc}$ and $\texttt{fd}_0, \ldots, \texttt{fd}_{\log_2 N-1}$ are updated incrementally. Once all leaf seeds have been processed, these accumulators are transformed into the desired output variables. The same incremental approach can be used to implement BLC.ConvertToLineEvaluation.

Algorithm 38 BLC.ConvertToLine($\mathtt{salt}, e, \mathtt{lseed}[e], x$)

Input: a salt $\mathtt{salt} \in \{0,1\}^\lambda$, an execution index $e \in [0, \tau-1]$, seed array $\mathtt{lseed}[e] \in (\{0,1\}^\lambda)^N$, an MQ secret solution x

Output: seed commitments $\mathtt{ls\_com}[e]$, an auxiliary value $\Delta_x \in \mathbb{F}^n$, coefficients x_0, u_0, u_1

$\mathtt{tweaked\_salt}_0 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 0, e)$
$\mathtt{tweaked\_salt}_1 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 1, e)$
$\mathtt{acc} = 0$ ▷ $\mathtt{acc} \in \{0,1\}^{n+\mu\cdot\eta}$
$\mathtt{fd}_0 = 0, \ldots, \mathtt{fd}_{\log_2 N - 1} = 0$ ▷ $\mathtt{fd}_j \in \{0,1\}^{n+\mu\cdot\eta}$
for $i = 0$ **to** $N-1$ **do**
    $\mathtt{ls\_com}[e][i] \leftarrow \mathsf{SeedCommit}(\mathtt{tweaked\_salt}_0, \mathtt{tweaked\_salt}_1, \mathtt{lseed}[e][i])$
    $\mathtt{raw}_i \leftarrow \mathsf{SeedExpand}(\mathtt{salt}, e, \mathtt{lseed}[e][i], |x|_8 + |u|_8)$
    $\mathtt{acc} \leftarrow \mathtt{acc} \oplus \mathtt{raw}_i$
    $\mathtt{fd}_{p_i} \leftarrow \mathtt{fd}_{p_i} \oplus \mathtt{acc}$ ▷ p_i is the position of the only bit set in $\omega_i \oplus \omega_{i+1}$
$(\bar{x}_{\text{ACC}}, \bar{u}_{\text{ACC}}) \leftarrow \mathsf{Parse}(\mathtt{acc})$ ▷ $\bar{x}_{\text{ACC}} \in \mathbb{F}^n$, $\bar{u}_{\text{ACC}} \in \mathbb{K}^\eta$
$\bar{x}_{\text{FOLD}} = 0, \bar{u}_{\text{FOLD}} = 0$ ▷ $\bar{x}_{\text{FOLD}} \in \mathbb{K}^n$, $\bar{u}_{\text{FOLD}} \in \mathbb{K}^\eta$
for $j = 0$ **to** $\log_2 N - 1$ **do**
    $(\mathtt{fd}^x_j, \mathtt{fd}^u_j) \leftarrow \mathsf{Parse}(\mathtt{fd}_j)$ ▷ $\mathtt{fd}^x_j \in \mathbb{F}^n$, $\mathtt{fd}^u_j \in \mathbb{K}^\eta$
    $\bar{x}_{\text{FOLD}} \leftarrow \bar{x}_{\text{FOLD}} + e_j \cdot \mathtt{fd}^x_j$ ▷ $\{e_0, \ldots, e_{\log_2 |\mathbb{K}|-1}\}$ is the canonical $\mathbb{F}_2$-basis of $\mathbb{K}$
    $\bar{u}_{\text{FOLD}} \leftarrow \bar{u}_{\text{FOLD}} + e_j \cdot \mathtt{fd}^u_j$
$u_0 \leftarrow -\bar{u}_{\text{FOLD}}$ ▷ $u_0 \in \mathbb{K}^\eta$
$u_1 \leftarrow \bar{u}_{\text{ACC}}$ ▷ $u_1 \in \mathbb{K}^\eta$
$x_0 \leftarrow -\bar{x}_{\text{FOLD}}$ ▷ $x_0 \in \mathbb{K}^n$
$\Delta_x[e] \leftarrow x - \bar{x}_{\text{ACC}}$ ▷ $\Delta_x \in \mathbb{F}^n$
return $(\mathtt{ls\_com}[e], \Delta_x[e], x_0, u_0, u_1)$

4.2 Low-memory implementation

Node recomputation. In the current description of the BLC routines, the GGM trees, as well as all leaf-seed commitments, are kept in memory. Between the commitment and opening phases, they are stored as part of the BLC opening key. However, these data structures are very large, ranging from a few hundred kilobytes to several megabytes depending on the parameter set. Consequently, an implementation directly following the pseudo-code of this specification would require a significant amount of memory. Table 18 reports the corresponding memory requirements for all MQOM parameter sets.

Table 18: Memory usage of a direct implementation of the BLC trees and commitments.

Instance	τ	N	Memory Usage		
			Nodes + Leaves	Commitments	Total
MQOM3-L1-gf2-shorter-ct	10	4096	2× 655 KB	1 311 KB	2 621 KB
MQOM3-L1-gf2-shorter-ot	10	8192	2× 1 311 KB	2 621 KB	5 242 KB
MQOM3-L1-gf16-short-ct/ot	12	2048	2× 393 KB	786 KB	1 573 KB
MQOM3-L1-gf16-fast-ct/ot	17	256	2× 70 KB	139 KB	279 KB
MQOM3-L3-gf2-shorter-ct	16	4096	2× 1 572 KB	3 146 KB	6 291 KB
MQOM3-L3-gf2-shorter-ot	17	4096	2× 1 671 KB	3 342 KB	6 685 KB
MQOM3-L3-gf16-short-ct	18	2048	2× 885 KB	1 769 KB	3 539 KB
MQOM3-L3-gf16-short-ot	19	2048	2× 934 KB	1 868 KB	3 736 KB
MQOM3-L3-gf16-fast-ct/ot	26	256	2× 160 KB	319 KB	639 KB
MQOM3-L5-gf2-shorter-ct/ot	22	4096	2× 2 884 KB	5 767 KB	11 534 KB
MQOM3-L5-gf16-short-ct/ot	25	2048	2× 1 638 KB	3 277 KB	6 554 KB
MQOM3-L5-gf16-fast-ct	35	256	2× 287 KB	573 KB	1 147 KB
MQOM3-L5-gf16-fast-ot	36	256	2× 295 KB	590 KB	1 180 KB

Fortunately, there is an alternative implementation strategy. For signing, instead of storing all these values in the commitment phase, they can be discarded and recomputed during the opening phase. The key observation is that only a small subset of the data needs to be recomputed: the sibling paths and the seed commitments corresponding to the hidden leaves. This can be done efficiently incurring a small overhead compared to the cost of generating the complete GGM trees and all leaf commitments.

We give in Algorithms 39 and 40 an alternative description of the routines CT.BLC.Open and CT.SmallGGMTree.Open for the correlated-tree variant. In this description, the BLC opening key `key` contains the salt, the master seed, the first λ bits δ of the MQ solution, and the auxiliary values $\Delta_x^{(1)}$. Rather than merely returning the sibling path, Algorithm 40 also outputs the hidden leaf seed. This allows the corresponding leaf commitment to be recomputed during the opening procedure, thereby avoiding the need to store all leaf commitments in the opening key.

We give in Algorithms 41 and 42 an alternative description of the routines OT.BLC.Open and OT.BigGGMTree.Open for the one-tree variant. In this description, the BLC opening key `key` contains the salt, the master seed, and the auxiliary values Δ_x.

For memory-constrained devices, this optimization is crucial. Storing all tree nodes and leaf commitments would simply be infeasible, whereas selectively recomputing the required values dramatically reduces the memory footprint with only a limited computational overhead.

Algorithm 39 CT.BLC.Open($\mathtt{key}, i^*, \alpha_1$)

Input: a commitment key $\mathtt{key}$, an index array i^*, τ vectors $\alpha_1[0], \ldots, \alpha_1[\tau-1] \in \mathbb{K}^\eta$
Output: a BLC opening $\mathtt{opening} \in \{0,1\}^{\lambda \cdot \tau \cdot (\log_2(N)+2)+\tau \cdot n \cdot \log_2(|\mathbb{F}|)}$

1: $(\mathtt{salt}, \mathtt{mseed}, \delta, \Delta_x^{(1)}) \leftarrow \mathsf{Parse}(\mathtt{key})$
2: **for** $e = 0$ **to** $\tau - 1$ **do**
3: $\quad(\mathtt{path}[e], \mathtt{hidden\_leaf}[e])$
4: $\quad\quad\leftarrow \mathsf{CT.SmallGGMTree.OpenWithoutTree}(\mathtt{salt}, \mathtt{mseed}, e, \delta, i^*[e])$
5: $\quad\mathtt{tweaked\_salt}_0 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 0, e)$
6: $\quad\mathtt{tweaked\_salt}_1 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 1, e)$
7: $\quad\mathtt{out\_ls\_com}[e] \leftarrow \mathsf{SeedCommit}(\mathtt{tweaked\_salt}_0, \mathtt{tweaked\_salt}_1, \mathtt{hidden\_leaf}[e])$
8: $\quad\mathtt{chunk}[e] \leftarrow \mathsf{Serialize}(\mathtt{path}[e], \mathtt{out\_ls\_com}[e], \Delta_x^{(1)}[e], \alpha_1[e])$
9: $\mathtt{opening} \leftarrow \mathsf{Serialize}(\mathtt{chunk})$
10: **return** $\mathtt{opening}$

Algorithm 40 CT.SmallGGMTree.OpenWithoutTree($\mathtt{salt}, \mathtt{mseed}, e, \delta, i^*[e]$)

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

1: $\mathtt{lidx} \leftarrow N + i^*[e]$
2: $\mathtt{parent} \leftarrow \mathtt{mseed}$
3: $\mathtt{hidden} \leftarrow \delta$
4: **for** $j = 0$ **to** $\log_2(N) - 1$ **do**
5: $\quad\mathtt{tweaked\_salt} = \mathsf{TweakSalt}(\mathtt{salt}, 2, \mathsf{IndexIdentifier}(e, j))$
6: $\quad\mathtt{left} \leftarrow \mathsf{SeedDerive}(\mathtt{tweaked\_salt}, \mathtt{parent})$
7: $\quad\mathtt{right} \leftarrow \mathtt{left} \oplus \mathtt{hidden}$
8: $\quad$**if** the $(\log_2(N) - j)$-th bit of $\mathtt{lidx}$ is 1 **then**
9: $\quad\quad\mathtt{path}[e][\log_2(N) - 1 - j] = \mathtt{left}$
10: $\quad\quad\mathtt{hidden} = \mathtt{right}$
11: $\quad$**else**
12: $\quad\quad\mathtt{path}[e][\log_2(N) - 1 - j] = \mathtt{right}$
13: $\quad\quad\mathtt{hidden} = \mathtt{left}$
14: $\quad\mathtt{parent} \leftarrow \mathtt{hidden}$
15: **return** $(\mathtt{path}, \mathtt{hidden})$

Algorithm 41 OT.BLC.Open($\mathtt{key}, i^*, \alpha_1$)

Input: a commitment key `key`, an index array i^*, τ vectors $\alpha_1[0], \ldots, \alpha_1[\tau-1] \in \mathbb{K}^\eta$
Output: a BLC opening `opening` $\in \{0,1\}^{3\lambda\cdot\tau+\lambda T_{\text{open}}+\tau\cdot n\cdot\log_2(|\mathbb{F}|)}$

1: $(\mathtt{salt}, \mathtt{mseed}, \Delta_x) \leftarrow \mathsf{Parse}(\mathtt{key})$
2: $\mathtt{hidden\_leaf\_idxs} = (i^*[0]\cdot\tau+0, \ldots, i^*[\tau-1]\cdot\tau+(\tau-1))$
3: $(\mathtt{path}, \mathtt{hidden\_leaves}) \leftarrow \mathsf{OT.BigGGMTree.OpenWithoutTree}(\mathtt{salt}, \mathtt{mseed}, \mathtt{hidden\_leaf\_idxs})$
4: **for** $e = 0$ **to** $\tau - 1$ **do**
5: $\quad \mathtt{tweaked\_salt}_0 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 0, e)$
6: $\quad \mathtt{tweaked\_salt}_1 \leftarrow \mathsf{TweakSalt}(\mathtt{salt}, 1, e)$
7: $\quad \mathtt{out\_ls\_com}[e]$
8: $\quad\quad \leftarrow \mathsf{SeedCommit}(\mathtt{tweaked\_salt}_0, \mathtt{tweaked\_salt}_1, \mathtt{hidden\_leaves}[i^*[e]\cdot\tau+e])$
9: $\mathtt{opening} \leftarrow \mathsf{Serialize}(\mathtt{path}, \mathtt{out\_ls\_com}, \Delta_x, \alpha_1)$
10: **return** `opening`

Algorithm 42 OT.BigGGMTree.OpenWithoutTree(`salt`, `mseed`, `hidden_leaf_idxs`)

Input: a salt `salt` $\in \{0,1\}^\lambda$, a master seed `mseed` $\in \{0,1\}^\lambda$, hidden leaf indexes `hidden_leaf_idxs`
Assumption: OT.BigGGMTree.IsValidOpeningSet(`hidden_leaf_idxs`) = **true**
Output: sibling path `path`

▷ Get indexes of the sensitive nodes
1: $\mathtt{hidden\_nodes} \leftarrow \mathsf{OT.GetSensitiveNodeIndexes}(\mathtt{hidden\_leaf\_idxs})$ ▷ Maximal list size: $\tau\cdot(h+1)$
2: $\mathtt{size} = |\mathtt{hidden\_nodes}| - 2\tau + 1$ ▷ Minimal opening size
▷ Collect the T_{open} nodes to be revealed
3: $\mathtt{path} = \emptyset$
4: $\mathtt{stack} \leftarrow \mathsf{Stack.Init}()$ ▷ Maximal stack size: $h+1$
5: $\mathtt{stack}.\mathsf{Push}((1, \mathtt{mseed}))$
6: $\mathtt{hidden\_leaves} \leftarrow \mathsf{Dict.Init}()$
7: **while** $\mathtt{stack}.\mathsf{GetLength}() > 0$ **do**
8: $\quad (k, \mathtt{node}) \leftarrow \mathtt{stack}.\mathsf{Pop}()$
9: $\quad$ **if** $k \notin \mathtt{hidden\_nodes}$ **then**
10: $\quad\quad$ **if** $\mathtt{size} < T_{\text{open}}$ and $\mathsf{IsLeaf}(k) =$ **false then** ▷ $\mathsf{IsLeaf}(k) =$ **false** *iff* $k < \tau\cdot N$
11: $\quad\quad\quad \mathtt{size} \leftarrow \mathtt{size} + 1$
12: $\quad\quad$ **else**
13: $\quad\quad\quad \mathtt{path} \leftarrow \mathtt{path} \,\|\, \mathtt{node}$
14: $\quad\quad\quad$ **continue**
15: $\quad$ **if** $\mathsf{IsLeaf}(k) =$ **false then**
16: $\quad\quad \mathtt{tweaked\_salt} = \mathsf{TweakSalt}(\mathtt{salt}, 2, k)$
17: $\quad\quad \mathtt{left\_child} \leftarrow \mathsf{SeedDerive}(\mathtt{tweaked\_salt}, \mathtt{node})$
18: $\quad\quad \mathtt{right\_child} \leftarrow \mathtt{left\_child} \oplus \mathtt{node}$
19: $\quad\quad \mathtt{stack}.\mathsf{Push}((2k+1, \mathtt{right\_child}))$
20: $\quad\quad \mathtt{stack}.\mathsf{Push}((2k, \mathtt{left\_child}))$
21: $\quad$ **else**
22: $\quad\quad \mathtt{hidden\_leaves}.\mathsf{Append}(\mathsf{key} = \mathsf{OT.FromLeafPosition}(k), \mathsf{value} = \mathtt{node})$
23: **return** $(\mathtt{path}, \mathtt{hidden\_leaves})$

Streamed expansion. On memory-constrained devices, the optimization described above is often insufficient, since storing all the leaf seeds already requires a significant amount of memory. Therefore, to achieve a low-memory implementation, one should rely on a streamed implementation of the tree expansion together with incremental processing of the leaf seeds. In particular, the folding and hashing computations should be performed incrementally as the leaf seeds are generated.

Implementing the tree expansion in a streaming fashion is straightforward: it simply requires traversing the tree in depth-first search (DFS) order. However, combining such a DFS traversal with a vectorized implementation of the block cipher may be challenging, although implementations supporting this approach exist [BF26]. Implementing a DFS traversal for partial tree expansion is slightly more involved, as the sibling path must also be handled appropriately. We give in Algorithm 43 a memory-efficient description of the routine OT.BigGGMTree.PartiallyExpand. The corresponding partial expansion for a small tree is simpler, since there is only one hidden leaf.

Algorithm 43 OT.BigGGMTree.PartiallyExpand($\texttt{salt}, \texttt{path}, \texttt{hleaves\_idxs}$)

Input: a salt $\texttt{salt} \in \{0,1\}^\lambda$, a sibling path `path`, hidden leaf indexes `hleaves_idxs`
Output: a partial leaf seed array `lseed_all`

▷ Get indexes of the sensitive nodes
1: $\texttt{hidden\_nodes} \leftarrow$ OT.GetSensitiveNodeIndexes($\texttt{hleaves\_idxs}$) ▷ Maximal list size: $\tau \cdot (h+1)$
2: $\texttt{size} = |\texttt{hidden\_nodes}| - 2\tau + 1$ ▷ Minimal opening size
3: **if** $\texttt{size} > T_{\text{open}}$ **then**
4: **return** $\perp$

▷ Recover the open leaves
5: $\texttt{stack} \leftarrow$ Stack.Init() ▷ Maximal stack size: $h+1$
6: stack.Push($(1, \emptyset)$)
7: **while** stack.GetLength() > 0 **do**
8: $(k, \texttt{node}) \leftarrow$ stack.Pop()
9: **if** $\texttt{node} = \emptyset$ and $k \notin \texttt{hidden\_nodes}$ **then**
10: **if** $\texttt{size} < T_{\text{open}}$ and IsLeaf(k) = **false then**
11: $\texttt{size} \leftarrow \texttt{size} + 1$
12: **else**
13: $\texttt{node} \,\|\, \texttt{path} \leftarrow \texttt{path}$ ▷ $\texttt{node} \in \{0,1\}^\lambda$
14: **if** IsLeaf(k) = **false then** ▷ IsLeaf(k) = **false** *iff* $k < \tau \cdot N$
15: $\texttt{left\_child} = \emptyset, \texttt{right\_child} = \emptyset$
16: **if** $\texttt{node} \neq \emptyset$ **then**
17: $\texttt{tweaked\_salt} =$ TweakSalt($\texttt{salt}, 2, k$)
18: $\texttt{left\_child} \leftarrow$ SeedDerive($\texttt{tweaked\_salt}, \texttt{node}$)
19: $\texttt{right\_child} \leftarrow \texttt{left\_child} \oplus \texttt{node}$
20: stack.Push($(2k+1, \texttt{right\_child})$)
21: stack.Push($(2k, \texttt{left\_child})$)
22: **else**
23: $\texttt{lseed\_all}[\text{OT.FromLeafPosition}(k)] = \texttt{node}$
24: **return** lseed_all

4.3 Precomputation

To reduce signing latency, most of the signing procedure can be precomputed before the message to be signed is available, provided that the signing key is already known. Indeed, most of the signing procedure is independent of the message: in particular, the BLC commitments and the PIOP computations can be performed in advance. The key observation is that the data that must be retained between this precomputation phase and the finalization of the signature are relatively small: for Category I instances, they range from approximately 250 to 900 bytes, which is significantly smaller than the signatures themselves. Consequently, even a memory-constrained device can store several such precomputed values, which we call *pre-signatures*.

We describe in Algorithm 44 the routine PreSign, which computes a pre-signature from a secret key. The algorithm essentially executes the message-independent instructions of the Sign routine, resulting in an unmasked pre-signature ($\texttt{presig\_id}, \texttt{salt}, \alpha_1, \texttt{key}$). For the BLC opening key, we use a compressed representation, denoted `ckey`, that contains only the master seed `mseed` and the auxiliary values $\Delta_x/\Delta_x^{(1)}$. Compared to the memory-optimized structure `key`, the salt is omitted and, for the correlated-tree variant, δ is omitted as well. The BLC routines for commitment and opening described in the presignature algorithms use the memory optimized representation of `key` introduced in the memory constrained implementation description (i.e. `key` is made of `salt`, `mseed`, along with δ and $\Delta_x^{(1)}$ for the correlated-tree variant, Δ_x for the one-tree variant).

Instead of storing this pre-signature directly, we first mask it. This precaution is important because a pre-signature may remain in memory for a relatively long time. In particular, an unmasked pre-signature allows the signing key to be recovered by re-expanding the trees and exploiting the auxiliary values. An unmasked pre-signature must therefore be considered *as sensitive as* the signing key itself. Masking allows the resulting pre-signature to be stored in untrusted memory. In Algorithm 44, we describe one possible masking mechanism, although an implementer may securely replace it with another one, since the masking procedure does not affect the resulting signature. To allow the mask to be recovered using only the secret key while ensuring that a fresh mask is used for every pre-signature, we sample a random bit string rnd and combine it with the secret key to derive the mask. We use a 32-byte value for rnd at all security levels, making collisions negligible even when up to 2^{64} signatures are generated under the same signing key. We deliberately do not use the signature salt for this purpose, for two reasons: first, its size of only λ bits would not provide a sufficiently small collision probability; second, using it would make it possible to link a stored pre-signature to the corresponding final signature without knowledge of the secret key. If linkability is not a concern, it is sufficient to mask only the master seed `mseed` contained in `key`, since all the other components of the pre-signature will eventually be revealed as part of the final signature.

We describe in Algorithm 45 the routine FinalizeSign, which transforms a pre-signature into a final signature once the message becomes available. The sizes of the stored pre-signatures are reported in Table 19. The compressed `ckey` can be easily expanded into `key` by adding the salt and, in the correlated-tree variant, deriving the missing δ from `sk`.

Special care must be taken when managing pre-signatures. In particular, reusing the same pre-signature to sign two different messages completely reveals the signing key. Implementations must therefore guarantee that each pre-signature is consumed at most once. When a stateful mechanism is available, rnd may alternatively be implemented as a counter. The signing system can then enforce that a pre-signature is accepted only if its rnd value is strictly larger than the last consumed value, thereby preventing pre-signature reuse.

Remark 7. *Precomputation is particularly attractive for side-channel-protected implementa-*

Algorithm 44 PreSign(sk)

Input: a secret key sk
Output: a pre-signature presig

▷ Start the signing operation
$(\mathtt{pk}, x) = \mathsf{Parse}(\mathtt{sk})$
$(\mathtt{mseed\_eq}, \hat{y}) = \mathsf{Parse}(\mathtt{pk})$
$\mathtt{mseed} \leftarrow \{0,1\}^{\lambda}$
$\mathtt{salt} \leftarrow \{0,1\}^{\lambda}$
$(\mathtt{com}_1, \mathtt{key}, x_0, u_0, u_1) \leftarrow \mathsf{BLC.Commit}(\mathtt{mseed}, \mathtt{salt}, x)$
$(\alpha_0, \alpha_1) \leftarrow \mathsf{ComputePAlpha}(\mathtt{com}_1, x_0, u_0, u_1, x, \mathtt{mseed\_eq})$
$\mathtt{com}_2 \leftarrow \mathsf{Hash}_1(\{(\alpha_0[e], \alpha_1[e])\}_e)$ ▷ $\mathsf{Hash}_1(\alpha_0[0], \alpha_1[0], \alpha_0[1], \ldots)$
$\mathtt{presig\_id} \leftarrow \mathsf{Hash}_2(\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2)$
▷ Compute a masked presigning material
$\mathsf{rnd} \leftarrow \{0,1\}^{256}$
$\mathtt{ckey} \leftarrow \mathtt{key}$ ▷ Drop salt, for CT: drop δ
$\mathsf{data} \leftarrow \mathsf{Serialize}(\mathtt{presig\_id}, \mathtt{salt}, \alpha_1, \mathtt{ckey})$
$\mathsf{mask} \leftarrow \mathsf{XOF}(\mathtt{sk} \,\|\, \mathsf{rnd}, 3\lambda + |\alpha_1| + |\mathtt{ckey}|)$
$\mathsf{masked\_data} \leftarrow \mathsf{data} \oplus \mathsf{mask}$
return $\mathsf{presig} := \mathsf{Serialize}(\mathsf{rnd}, \mathsf{masked\_data})$

Algorithm 45 FinalizeSign(sk, presig, msg)

Input: a secret key sk, a pre-signature presig, a message msg
Output: a signature sig

▷ Recover the signing elements
$(\mathsf{rnd}, \mathsf{masked\_data}) = \mathsf{Parse}(\mathsf{presig})$
$\mathsf{mask} \leftarrow \mathsf{XOF}(\mathtt{sk} \,\|\, \mathsf{rnd}, 3\lambda + |\alpha_1| + |\mathtt{ckey}|)$
$\mathsf{data} \leftarrow \mathsf{masked\_data} \oplus \mathsf{mask}$
$(\mathtt{presig\_id}, \mathtt{salt}, \alpha_1, \mathtt{ckey}) = \mathsf{Parse}(\mathsf{data})$
$\mathtt{key} \leftarrow \mathtt{ckey}$ ▷ salt recovered from previous parsing, for CT: δ computed from sk
▷ Finalize the signing operation using the message
$\mathtt{msg\_hash} \leftarrow \mathsf{Hash}_3(\mathtt{msg})$
$\mathtt{sig\_id} \leftarrow \mathsf{Hash}_4(\mathtt{presig\_id}, \mathtt{msg\_hash})$
$(i^*, \mathtt{nonce}) \leftarrow \mathsf{SampleChallenge}(\mathtt{sig\_id})$
if $(i^*, \mathtt{nonce}) = \perp$, **return** $\perp$
$\mathtt{opening} \leftarrow \mathsf{BLC.Open}(\mathtt{key}, i^*, \alpha_1)$
return $\mathsf{sig} := \mathsf{Serialize}(\mathtt{sig\_id}, \mathtt{salt}, \mathtt{nonce}, \mathtt{opening})$

tions. In the online FinalizeSign *routine, two operations may have a non-negligible computational cost: the grinding procedure and the tree-path opening. However, the grinding procedure manipulates only public values and therefore does not need to be protected against side-channel leakage. Consequently, only the path-opening computation requires protection, mitigating the impact of grinding on protected implementations, even for instances where its computational cost is significant.*

Table 19: Sizes of the pre-signatures. Since rnd and its size are implementation choices, |rnd| (32 bytes here) are not included and must be added to the figures.

Instance	Sig. Size	Pre-sig. Size
MQOM3-L1-gf2-shorter-ct	2 492 B	264 B
MQOM3-L1-gf2-shorter-ot	2 492 B	424 B
MQOM3-L1-gf16-short-ct	2 932 B	448 B
MQOM3-L1-gf16-short-ot	2 932 B	640 B
MQOM3-L1-gf16-fast-ct	3 316 B	608 B
MQOM3-L1-gf16-fast-ot	3 316 B	880 B
MQOM3-L3-gf2-shorter-ct	5 932 B	576 B
MQOM3-L3-gf2-shorter-ot	5 890 B	1 014 B
MQOM3-L3-gf16-short-ct	6 556 B	960 B
MQOM3-L3-gf16-short-ot	6 556 B	1 464 B
MQOM3-L3-gf16-fast-ct	7 564 B	1 344 B
MQOM3-L3-gf16-fast-ot	7 660 B	1 968 B
MQOM3-L5-gf2-shorter-ct	10 836 B	1 008 B
MQOM3-L5-gf2-shorter-ot	10 804 B	1 712 B
MQOM3-L5-gf16-short-ct	12 100 B	1 728 B
MQOM3-L5-gf16-short-ot	11 716 B	2 528 B
MQOM3-L5-gf16-fast-ct	13 540 B	2 368 B
MQOM3-L5-gf16-fast-ot	13 380 B	3 584 B

4.4 Signature streaming

Streamed signing. Except for the salt and the signature identifier (representing 3λ bits in total), all signature components are derived only toward the end of the signing algorithm. Therefore, signature streaming cannot be used to significantly reduce the signing latency. Nevertheless, it can still be useful for reducing the memory footprint of the signing procedure.

For the correlated-tree variant, the signature transcript can be streamed on a per-repetition basis: the data associated with each parallel repetition can be output as soon as they have been computed, since these data are ordered by repetition in the signature transcript.

For the one-tree variant, the largest signature component is the sibling path, containing T_{open} nodes. Thanks to the ordering of these nodes, they can be output incrementally as soon as they are computed, without storing the entire sibling path in memory.

Streamed verification. The one-tree variant is less amenable to streaming than the correlated-tree variant because the sibling paths associated with the different parallel repetitions are interleaved.

For the correlated-tree variant, the verifier can process an entire parallel repetition as soon as the corresponding signature data have been received. There is no need to wait for the components associated with the other repetitions, except for the salt and the signature identifier, which appear at the beginning of the signature transcript. This enables verification with a very small memory footprint. More generally, this means that the repetitions can be verified independently, in any order or in parallel [BF26]. Therefore, the verification is highly parallelizable and each repetition data can be sent in a different communication thread (*e.g.* IP packet)

out-of-order and the verification can start before all the data has been received. Let us stress that the 2λ first bits of the signature, denoted `sig_id`, can be used as a signature identifier. Knowing two signatures with the same `sig_id` would imply knowing a SHAKE hash collision.

For the one-tree variant, the verifier could process the nodes of the sibling path incrementally as they are received, without storing the entire path. However, doing so would require processing the τ parallel repetitions simultaneously, resulting in a larger memory footprint. Moreover, the arithmetic computations (*i.e.* the PIOP routines) cannot start before the entire sibling path has been received and processed.

4.5 Worst-case execution

The overall structure of MQOM makes its running time constant by design. The only exception is the grinding procedure in SampleChallenge, which is used solely to reduce the signature size. More precisely, SampleChallenge tests up to 2^{32} nonce values until finding one that yields a valid opening challenge. Grinding is parameterized so that the signer must perform an average of 2^w AES (or Rijndael) computations to find a valid nonce, for some parameter w. The one-tree variant further uses a rejection mechanism to ensure that the sibling path size is at most T_{open}. The total average number of AES computations to find a valid challenge is therefore $2^{w_{\text{tot}}}$, with

$$w_{\text{tot}} = \begin{cases} w & \text{for the correlated-tree variant,} \\ w + (-\log_2(p_{\text{rej}})) & \text{for the one-tree variant,} \end{cases}$$

where $p_{\text{rej}} = \Pr[\text{“size of sibling path} \leq T_{\text{open}}\text{”}]$ is the probability that a challenge is rejected in the one-tree variant. The values of w_{tot} for all instances are given in Table 20.

Within grinding, each nonce test roughly consists of two AES/Rijndael encryptions (excluding the key schedule, which can be factorized across tests) where each nonce is accepted with probability $2^{-(w-1)}$. The repeated computation is therefore very small and independent of secret values, so the resulting timing variations do not introduce timing leakage on secret data. Moreover, for any target failure probability, we can easily derive a bound on the number of nonce values that need to be tested. Table 20 gives this bound for all instances and for failure probabilities of 2^{-32}, 2^{-48}, and 2^{-64}. For instance, for MQOM3-L1-gf16-fast-ct, testing at most 11 335 nonce values is sufficient to achieve a failure probability below 2^{-64}. Let us stress that each iteration involves only two AES/Rijndael encryptions, plus, in the one-tree variant, potentially some hashing and tree-traversal operations.

For the correlated-tree variant, we can easily estimate the overhead of such a bounded worst-case execution compared with the average-case execution. To do so, we account for the two encryptions performed by each grinding iteration and compare them with the total number of encryptions performed by the signing algorithm, including those used for tree derivation, seed commitments, and seed expansion. Table 21 reports the resulting upper bound on the ratio between the bounded worst-case and average-case execution times. For instance, for MQOM3-L1-gf16-fast-ct, targeting a failure probability of 2^{-64} yields a worst-case execution time of at most 1.41 times the average-case execution time (i.e., an overhead of 41%). Excluding the shorter instances, which make aggressive use of grinding to reduce signature sizes, this factor remains below 2 for all instances. Let us emphasize that these estimates are conservative upper bounds, as they completely neglect the arithmetic and hashing computations performed by the rest of the signing procedure. For the one-tree variant, deriving similarly implementation-independent estimates is more difficult, since each grinding iteration may involve additional and more expensive computations.

Table 20: Repetition counts for the grinding procedure of MQOM.

Parameter Sets	Rej. Parameters			Grinding repetition counts		
	w	T_{open}	w_{tot}	2^{-32}	2^{-48}	2^{-64}
MQOM3-L1-gf2-shorter-ct	18	-	18.0	2 907 259	4 360 889	5 814 518
MQOM3-L1-gf2-shorter-ot	8	110	14.7	295 169	442 753	590 338
MQOM3-L1-gf16-short-ct	8	-	8.0	2 829	4 243	5 657
MQOM3-L1-gf16-short-ot	8	120	9.6	8 596	12 894	17 192
MQOM3-L1-gf16-fast-ct	9	-	9.0	5 668	8 501	11 335
MQOM3-L1-gf16-fast-ot	9	119	10.8	19 762	29 643	39 524
MQOM3-L3-gf2-shorter-ct	16	-	16.0	726 807	1 090 210	1 453 613
MQOM3-L3-gf2-shorter-ot	5	170	14.5	256 958	385 437	513 916
MQOM3-L3-gf16-short-ct	12	-	12.0	45 416	68 123	90 831
MQOM3-L3-gf16-short-ot	2	175	10.0	11 346	17 019	22 691
MQOM3-L3-gf16-fast-ct	10	-	10.0	11 346	17 019	22 691
MQOM3-L3-gf16-fast-ot	10	186	11.0	22 702	34 053	45 404
MQOM3-L5-gf2-shorter-ct	14	-	14.0	181 694	272 540	363 387
MQOM3-L5-gf2-shorter-ot	14	241	16.0	726 807	1 090 210	1 453 613
MQOM3-L5-gf16-short-ct	6	-	6.0	699	1 048	1 398
MQOM3-L5-gf16-short-ot	6	238	11.9	42 373	63 560	84 746
MQOM3-L5-gf16-fast-ct	11	-	11.0	22 702	34 053	45 404
MQOM3-L5-gf16-fast-ot	4	235	11.7	36 887	55 330	73 773

Table 21: Upper-bounds of the worst-case factors for the correlated-tree instances of MQOM.

Parameter Sets	Grinding repetition counts		
	2^{-32}	2^{-48}	2^{-64}
MQOM3-L1-gf2-shorter-ct	≤ 6.2	≤ 9.1	≤ 11.9
MQOM3-L1-gf16-short-ct	≤ 1.02	≤ 1.03	≤ 1.04
MQOM3-L1-gf16-fast-ct	≤ 1.19	≤ 1.30	≤ 1.41
MQOM3-L3-gf2-shorter-ct	≤ 2.4	≤ 3.2	≤ 4.0
MQOM3-L3-gf16-short-ct	≤ 1.18	≤ 1.28	≤ 1.39
MQOM3-L3-gf16-fast-ct	≤ 1.25	≤ 1.39	≤ 1.53
MQOM3-L5-gf2-shorter-ct	≤ 1.30	≤ 1.46	≤ 1.62
MQOM3-L5-gf16-short-ct	≤ 1.00	≤ 1.00	≤ 1.00
MQOM3-L5-gf16-fast-ct	≤ 1.37	≤ 1.57	≤ 1.78

Finally, let us emphasize that grinding affects only signing, not signature verification. Alternative design choices could avoid the need to consider a probabilistic worst-case execution time. The simplest approach would be to disable grinding, at the cost of slightly larger signatures. Another possibility would be to use a deterministic symmetric grinding procedure that does not rely on a stochastic event. However, in this case, verification would incur an additional performance cost, since the verifier would need to repeat the same grinding computation as the signer.

Remark 8. *The grinding procedure only involves public values and therefore does not require*

side-channel protection. Consequently, its relative impact on the worst-case execution time is even smaller when considering an SCA-protected implementation.

5 Security

5.1 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 $\texttt{pk}$ 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 $\texttt{msg}$ is not one of requests to the signature oracle and such that σ is a valid signature of $\texttt{msg}$ with respect to $\texttt{pk}$.

Partial Guessing One-wayness of MQ. As analyzed in [FR26; KX26], the EUF-CMA security of the correlated-tree instances relies on a slightly weaker variant of the MQ problem, name the *partial guessing one-wayness* of the multivariate quadratic problem (PGOW-MQ).[4] Informally, the PGOW problem consists in finding the $k := \lambda / \log_2 |\mathbb{F}|$ first coordinates of the MQ solution given access to a validation oracle, which determines whether a candidate string matches those first k coordinates.

Definition 2 (PGOW-MQ Problem)**.** *Let $\mathbb{F}$ be a finite field of characteristic 2 and let m, n, λ be positive integers such that $\log_2 |\mathbb{F}|$ divides λ. The Partial Guessing One-wayness of Multivariate Quadratic problem (PGOW-MQ) with parameters $(\lambda, \mathbb{F}, m, n)$ 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 $y_i := x^\intercal A_i x + b_i^\intercal x$.*

Let $k := \lambda / \log_2 |\mathbb{F}|$, and let $\mathcal{O}^x$ be an oracle that, on input $\hat{x} \in \mathbb{F}^k$, outputs $\top$ if $\hat{x} = (x_1, \ldots, x_k)$, and output $\perp$ otherwise.

From $\big(\{A_i\}, \{b_i\}, y\big)$ and oracle access to $\mathcal{O}^x$, recover $\hat{x} \in \mathbb{F}^k$ such that $\mathcal{O}^x(\hat{x}) = \top$.

On the other hand, the EUF-CMA security of the one-tree instances relies on the hardness of the standard MQ problem, which is recalled in Definition 1.

Let us first note that MQ is strictly harder than PGOW-MQ: any algorithm that breaks MQ immediately yields an algorithm that breaks PGOW-MQ, without making any calls to the validation oracle. Conversely, in the absence of the validation oracle, PGOW-MQ could be tightly reduced to the standard MQ problem. Access to this oracle, however, introduces a reduction gap, making PGOW-MQ potentially weaker than MQ. As argued in Section 5.3, it seems unlikely that access to the validation oracle causes any significant security loss in practice, suggesting that PGOW-MQ is essentially as hard as MQ.

[4] Specifically, the PGOW notion was introduced in [FR26] to obtain a tight security reduction for the correlated-tree technique. The analysis of [KX26] considers a simpler variant of this problem, called *partial-domain one-wayness*, which is strictly stronger than PGOW but does not yield a tight security reduction.

Security assumptions. Our security statement is based on the following assumptions:

- **MQ hardness (one-tree variant).** Solving the considered MQ instance is $(t, \epsilon_{\text{MQ}})$-*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.
- **PGOW-MQ hardness (correlated-tree variant).** Solving the considered PGOW-MQ instance is $(t, Q, \epsilon_{\text{PGOW-MQ}})$-*hard* for some $(t, Q, \epsilon_{\text{PGOW-MQ}})$ which are implicit functions of the security parameter λ, where Q is the number of queries to the validation oracle. Formally, any adversary $\mathcal{A}$ on input a random MQ instance $\big(\{A_i\}, \{b_i\}, y\big)$ and access to the validation oracle, and running in time at most t while making at most Q oracle calls, has probability at most $\epsilon_{\text{PGOW-MQ}}$ to output the solution $\hat{x}$ of the input instance.
- **Random Oracle Model (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).

Key-only unforgeability. We first establish the security against key-only attacks (EUF-KO). This analysis applies identically to both variants (correlated-tree and one-tree), and its security reduces directly to the hardness of solving the MQ problem. In particular, it covers the use of independent batching challenges as well as the AES-based grinding optimization.

Theorem 5.1. *Let* $\{\mathsf{XOF}_i\}, \{\mathsf{Hash}_i\}$ *be modeled as a random oracles and* Enc *as an ideal cipher. Let* $\mathcal{A}$ *be an adversary against the EUF-KO security of* MQOM *running in time* t *and making a total of* Q_{RO} *queries to* XOF, Q_{IC} *queries to* Enc *and* Q_{Sign} *queries to the signing oracle, with* $Q_{Sign} \leq 2^{64}$, $Q_{RO} \leq 2^{\lambda}$, *and* $Q_{IC} \leq 2^{\lambda-1}$. *Assuming that* $\left(\frac{2}{N}\right)^{\tau} \leq \frac{2^{-72}}{\lambda}$, *the advantage of* $\mathcal{A}$ *in the EUF-KO game is upper-bounded as*

$$\begin{aligned}\mathsf{Adv}_{\text{EUF-KO}} \leq \epsilon_{\text{MQ}} + Q_{RO} \cdot \frac{1}{|\mathbb{K}|^{\eta}} + 2^{-w}\left(\frac{2}{N}\right)^{\tau} \cdot \frac{4 \cdot Q_{IC}}{3} \\ + \frac{3 \cdot Q_{RO}^2}{2^{2\lambda}} + \frac{(4N+1) \cdot Q_{RO} \cdot Q_{IC}}{2^{2\lambda}} + 2^{-6} \cdot \frac{Q_{IC}}{2^{\lambda}} + 2^{-68} \cdot \frac{Q_{RO}}{2^{\lambda}} + 2^{-\lambda}\end{aligned}$$

where ϵ_{MQ} *denotes the success probability of the algorithm for solving the MQ problem.*

Remark 9. *The condition* $Q_{IC} \leq 2^{\lambda-1}$ *can be relaxed: it is sufficient to require that the number of ideal-cipher queries made under any fixed key is at most* $2^{\lambda-1}$.

Proof. For the sake of simplicity, we assume that the secret key `sk` and the public key `pk` correspond to a truly random MQ instance (and not a pseudorandom instance from a master seed). Assume an EUF-KO adversary $\mathcal{A}$ exists which runs in time t and outputs a valid forgery with probability $\epsilon_{\text{EUF-KO}}$. We construct an algorithm $\mathcal{R}$ (the reduction) solving MQ in time t and probability ϵ_{MQ} from $\mathcal{A}$. Without loss of generality, we assume that $\mathcal{A}$ makes the queries to the random oracles and the ideal cipher which are necessary to verify the forged signature.

Otherwise, $\mathcal{A}$ can be modified to make these queries without significantly affecting its overall timing or success probability.

The reduction $\mathcal{R}$ receives an MQ challenge $\big(\{A_i\}, \{b_i\}, y\big)$ and aims to find the solution x. It interacts with $\mathcal{A}$ by simulating the EUF-KO game in which the public key is defined as the MQ challenge $pk := \big(\{A_i\}, \{b_i\}, y\big)$. The reduction answers the Hash (resp. XOF) queries of $\mathcal{A}$ with random values and maintains lists $\{\mathcal{Q}_{\mathsf{Hash}_i}\}_i$ (resp. $\{\mathcal{Q}_{\mathsf{XOF}_i}\}_i$) such that $(q, h) \in \mathcal{Q}_{\mathsf{Hash}_i}$ (resp. $(q, h) \in \mathcal{Q}_{\mathsf{XOF}_i}$) iff q has been queried to Hash_i (resp. XOF_i) and the random value h was picked and answered to this query. Similarly, the reduction answers the Enc queries of $\mathcal{A}$ with random values (keeping the permutation structure) and maintains lists $\mathcal{Q}_{\mathsf{Enc}[\mathtt{key}]}$ such that $(\mathtt{ptx}, \mathtt{ctx}) \in \mathcal{Q}_{\mathsf{Enc}[\mathtt{key}]}$ iff $(\mathtt{key}, \mathtt{ptx})$ has been queried to Enc and the random value $\mathtt{ctx}$ was picked and answered to this query (or if $(\mathtt{key}, \mathtt{ctx})$ has been queried to Enc^{-1} and the random value $\mathtt{ptx}$ was picked and answered to this query).

$\mathcal{R}$ maintains a list Bad of "bad answers" to hash queries. Whenever a random oracle is queried on a new input q, a random answer h is sampled. If $h \in \mathsf{Bad}$, then $\mathcal{R}$ aborts, otherwise $\mathcal{R}$ adds h to Bad. Additionally:

- for any query
$$q_2 = (\mathtt{pk}, \mathtt{com}_1, \mathtt{com}_2) \ ,$$
to Hash_2, $\mathcal{R}$ adds $\mathtt{com}_1$ and $\mathtt{com}_2$ to Bad;
- for any query
$$q_4 = (\mathtt{pk}, \mathtt{presig\_id}, \mathtt{msg\_hash}) \ ,$$
to Hash_4, $\mathcal{R}$ adds $\mathtt{presig\_id}$ and $\mathtt{msg\_hash}$ to Bad.

$\mathcal{R}$ also maintains some lists $\{\overline{\mathsf{Bad}}^{[\mathtt{tw\_salt}]}\}_{\mathtt{tw\_salt}}$ of "bad answers" for seed commitments. These bad answers are pairs of ciphertexts forming potential seed commitments. For any query

$$q_7 = (e, \mathtt{salt}, \mathtt{ls\_com}[e], \ldots)$$

to Hash_6, $\mathcal{R}$ adds the N commitment digests $\mathtt{ls\_com}[e][0], \ldots, \mathtt{ls\_com}[e][N-1]$ (each of them being a pair of ciphertexts) to the list $\overline{\mathsf{Bad}}^{[\mathtt{tw\_salt}]}$ for $\mathtt{tw\_salt} := \mathsf{TweakSalt}(\mathtt{salt}, 0, e)$. Whenever an ideal cipher is queried on a new plaintext $\mathtt{ptx}$ (resp. ciphertext $\mathtt{ctx}$) for a key $\mathtt{tw\_salt} := \mathsf{TweakSalt}(\mathtt{salt}, \mathsf{sel}, e)$ with $\mathsf{sel} \in \{0, 1\}$, a random answer $\mathtt{ctx}$ (resp. $\mathtt{ptx}$) is sampled while keeping the permutation structure. At the same time, $\mathcal{R}$ also samples a random ciphertext to the same plaintext $\mathtt{ptx}$ for the alternative key $\mathsf{TweakSalt}(\mathtt{salt}, \mathsf{sel} \oplus 1, e)$. If the obtained pair of ciphertexts collides with a digest from $\overline{\mathsf{Bad}}^{[\mathsf{TweakSalt}(\mathtt{salt},0,e)]}$, $\mathcal{R}$ aborts.

Then we get:

$$\begin{aligned}
\Pr[\mathcal{R} \text{ aborts because of } \mathsf{Bad}] &= (\#\text{ times a digest is sampled}) \cdot \Pr[\mathcal{R} \text{ aborts at that digest}] \\
&\leq Q_{\mathsf{RO}} \cdot \frac{\max |\mathsf{Bad}|}{2^{2\lambda}} \\
&= Q_{\mathsf{RO}} \cdot \frac{3 \cdot Q_2 + Q_3 + 3 \cdot Q_4 + Q_6 + Q_9}{2^{2\lambda}} \\
&\leq \frac{3 \cdot Q_{\mathsf{RO}}^2}{2^{2\lambda}} \qquad (14)
\end{aligned}$$

and, assuming that the maximal number of cipher calls with the *same* key is *at most* $2^{\lambda-1}$, we get:

$$\begin{aligned}\Pr[\mathcal{R} \text{ aborts because of } \overline{\mathsf{Bad}}] &= (\#\text{ times a digest is sampled}) \cdot \Pr[\mathcal{R} \text{ aborts at that digest}] \\ &\leq Q_{\mathsf{IC}} \cdot \frac{\max|\overline{\mathsf{Bad}}|}{(2^{\lambda} - 2^{\lambda-1})^2} \\ &= Q_{\mathsf{IC}} \cdot \frac{N \cdot Q_7}{\frac{1}{4} \cdot 2^{2\lambda}} \\ &\leq \frac{4N \cdot Q_{\mathsf{RO}} \cdot Q_{\mathsf{IC}}}{2^{2\lambda}} \end{aligned} \tag{15}$$

Therefore, combining (14) and (15), we get

$$\Pr[\mathcal{R} \text{ aborts}] \leq \frac{3 \cdot Q_{\mathsf{RO}}^2}{2^{2\lambda}} + \frac{4N \cdot Q_{\mathsf{RO}} \cdot Q_{\mathsf{IC}}}{2^{2\lambda}} \ . \tag{16}$$

In addition, $\mathcal{R}$ maintains an associative array $\mathcal{T}_{\mathsf{x}}$ to keep track of the candidate witnesses it gets from $\mathcal{A}$'s queries. Specifically, for each query

$$q_7 = (e, \mathtt{salt}, \mathtt{ls\_com}[e], \Delta_x)$$

to Hash_7, $\mathcal{R}$ checks whether for all $i \in [0 : N-1]$, there exists a seed $\mathtt{lseed}[e][i]$ such that

$$\mathtt{ls\_com}[e][i] = \mathsf{SeedCommit}(\mathsf{TweakSalt}(\mathtt{salt}, 0, e), \mathsf{TweakSalt}(\mathtt{salt}, 1, e), \mathtt{lseed}[e][i])$$

using queries from $\mathcal{Q}_{\mathsf{Enc}}$. If so, $\mathcal{R}$ defines $(\bar{x}_i, \bar{u}_i) := \mathsf{SeedExpand}(\mathtt{salt}, e, \mathtt{lseed}[e][i])$ for all $i \in [0 : N-1]$, and it defines

$$\mathcal{T}_{\mathsf{x}}[q_7] := \Delta_x + \sum_{i=0}^{N-1} \bar{x}_i \ .$$

Upon reception of a valid message-signature pair from $\mathcal{A}$, $\mathcal{R}$ checks $\mathcal{T}_{\mathsf{x}}$ for possible candidate solutions to the MQ challenge. If one of the $\mathcal{T}_{\mathsf{x}}[q_7]$ stores a correct solution x to the MQ challenge $\big(\{A_i\}, \{b_i\}, y\big)$, then $\mathcal{R}$ returns x, otherwise $\mathcal{R}$ returns $\perp$. We have

$$\begin{aligned}\Pr[\mathcal{A} \text{ wins}] &= \Pr[\mathcal{A} \text{ wins} \wedge \mathcal{R} \text{ aborts}] + \Pr[\mathcal{A} \text{ wins} \wedge \mathcal{R} \text{ outputs } \perp] + \Pr[\mathcal{A} \text{ wins} \wedge \mathcal{R} \text{ outputs } x] \\ &\leq \Pr[\mathcal{R} \text{ aborts}] + \Pr[\mathcal{A} \text{ wins} \mid \mathcal{R} \text{ outputs } \perp] + \Pr[\mathcal{R} \text{ outputs } x]\end{aligned}$$

which is

$$\epsilon_{\text{EUF-KO}} \leq \Pr[\mathcal{R} \text{ aborts}] + \Pr[\mathcal{A} \text{ wins} \mid \mathcal{R} \text{ outputs } \perp] + \epsilon_{\text{MQ}} \ . \tag{17}$$

$\Pr[\mathcal{R} \text{ aborts}]$ is upper bounded by Equation 16. It remains to upper bound $\Pr[\mathcal{A} \text{ wins} \mid \mathcal{R} \text{ outputs } \perp]$.

We now analyze the probability of $\mathcal{A}$ winning the EUF-KO game conditioned on the event that $\mathcal{R}$ outputs $\perp$, *i.e.*, that no suitable solution x was found from the hash queries.

Cheating in the first round. For any query $(q_7, h_7) \in \mathcal{Q}_{\mathsf{Hash}_7}$, and its corresponding expanded answer $\Gamma = \mathsf{XOF}_8(h_7)$, let us quantify the probability that $\mathcal{T}_{\mathsf{x}}[q_7] = x$ is non-empty and $\Gamma \cdot \hat{F}(x) = 0$ holds for x and Γ (this event is named the *bad first-round event*). Since $\mathcal{R}$ outputs $\perp$, x is not the right solution to the MQ challenge. Then by the uniformity of the Γ, the probability that it occurs is at most $\frac{1}{|\mathbb{K}|^{\eta}}$. By union event, the probability that it occurs for one query $(6, \cdot, \cdot)$ of $\mathcal{Q}_{\mathsf{Hash}}$ is at most

$$Q_{\mathsf{RO}} \cdot \frac{1}{|\mathbb{K}|^{\eta}} \ .$$

Cheating in the second round. Let us assume that no query raises the bad first-round event. Each hash query

$$q_6 = (\texttt{sig\_id}, \texttt{nonce}, c_0, c_1)$$

that $\mathcal{A}$ makes to Hash_6 can only be used in a winning signature if

- there exists a hash query $q_4 = (\texttt{pk}, \texttt{presig\_id}, \texttt{msg\_hash})$ to Hash_7 giving the hash digest $\texttt{sig\_id}$,
- there exists a hash query $q_2 = (\texttt{pk}, \texttt{com}_1, \texttt{com}_2)$ to Hash_2 giving the hash digest $\texttt{presig\_id}$,
- there exists τ corresponding queries $q_7[e] = (e, \texttt{salt}, \texttt{ls\_com}[e], \ldots)$ to Hash_7 giving the hash digest $\texttt{com}_1[e]$, with $e \in [0 : \tau - 1]$.

Then for each e, either the PIOP protocol failed, in which case $\mathcal{A}$ could not have won, or the PIOP protocol passed, despite $\Gamma^{(e)} \cdot \hat{F}(x^{(e)}) \neq 0$. Knowing that $D^e := P_u^e + \hat{F}(P^e) - P_\alpha^e$ is a non-zero degree-2 polynomial because no bad first-round event occurs, this implies that the PIOP evaluation point is among the roots of D^e (which are at most 2). Since the second-round challenge $\{i^{*[e]}\}_{e\in[1:\tau]} = \mathsf{XOF}_6(\texttt{sig\_id}, \texttt{nonce}, c_0, c_1)$ is distributed uniformly at random, the probability that this happens for all such bad second-round executions e is at most

$$\varepsilon_{\text{PIOP}} := \left(\frac{2}{N}\right)^{\tau} .$$

The probability that the adversary find such a successful query while passing the grinding check test with counter $\texttt{nonce}$ is given by the security analysis of the grinding construction. [Riv26, Theorem 1] analyzes the used AES-based grinding procedure and states that this probability is at most

$$\varepsilon_{\text{PIOP}} \cdot \left(\frac{4}{3}\frac{Q_{\mathsf{IC}}}{2^w} + 2^{64}\lambda \cdot \frac{Q_{\mathsf{IC}}}{2^\lambda} + 2^{64}\lambda \cdot \frac{Q_{\mathsf{IC}}}{2^\lambda - Q_{\mathsf{IC}}} + 2^{68}\lambda \cdot \frac{Q_{\mathsf{IC}}Q_{\mathsf{RO}}^3}{2^{4\lambda}} + \frac{2^{66}}{2^\lambda} \cdot \frac{Q_{\mathsf{IC}}Q_{\mathsf{RO}}^2}{2^{3\lambda}} + 10 \cdot \frac{Q_{\mathsf{RO}}}{2^\lambda}\right) + 2^{-(\lambda+1)} ,$$

while the last term is an upper bound to have $(\lambda+1)$-fold collision in $2Q_{\mathsf{RO}}$ uniform $(\lambda-2)$-bit values (when $\lambda \geq 128$). Using the fact that $\varepsilon_{\text{PIOP}} \leq \frac{2^{-(68+4)}}{\lambda}$, $Q_{\mathsf{IC}} \leq 2^{\lambda-1}$, $Q_{\mathsf{RO}} \leq 2^\lambda$, the above probability is at most

$$\varepsilon_{\text{PIOP}} \cdot \frac{4}{3}\frac{Q_{\mathsf{IC}}}{2^w} + 2^{-6} \cdot \frac{Q_{\mathsf{IC}}}{2^\lambda} + 2^{-4} \cdot \frac{Q_{\mathsf{IC}}Q_{\mathsf{RO}}}{2^{2\lambda}} + 2^{-68} \cdot \frac{Q_{\mathsf{RO}}}{2^\lambda} + 2^{-\lambda} . \tag{18}$$

Wrapping up. From Equation 16, Equation 17 and Equation 18, we finally get

$$\epsilon_{\text{EUF-KO}} \leq \frac{3 \cdot Q_{\mathsf{RO}}^2}{2^{2\lambda}} + \frac{4N \cdot Q_{\mathsf{RO}} \cdot Q_{\mathsf{IC}}}{2^{2\lambda}} + \epsilon_{\text{MQ}} + Q_{\mathsf{RO}} \cdot \frac{1}{|\mathbb{K}|^\eta} + \left(\frac{2}{N}\right)^{\tau} \cdot \frac{4}{3}\frac{Q_{\mathsf{IC}}}{2^w} + 2^{-6} \cdot \frac{Q_{\mathsf{IC}}}{2^\lambda} + 2^{-4} \cdot \frac{Q_{\mathsf{IC}}Q_{\mathsf{RO}}}{2^{2\lambda}} + 2^{-68} \cdot \frac{Q_{\mathsf{RO}}}{2^\lambda} + 2^{-\lambda} . \tag{19}$$

□

Unforgeability against chosen message attacks. EUF-CMA security is then established via a reduction to EUF-KO, relying on the fact that the signing oracle can be simulated without knowledge of the private key by programming the random oracles. For the correlated-tree variant, the reduction is given in [FR26] and additionally relies on the hardness of the PGOW-MQ problem. By contrast, for the one-tree variant, the reduction is simpler and relies solely on the hardness of the MQ problem, without requiring any additional computational assumption.

Salt collisions. The signature salt provides domain separation between different signatures. However, its size is only λ bits, and therefore salt collisions cannot be assumed to occur with negligible probability over the full number of allowed signature queries. Moreover, part of the salt is reserved for domain separation between encryption calls within a single signature (see the TweakSalt routine). More precisely, 16 bits are reserved in the correlated-tree variant and 24 bits in the one-tree variant, further reducing the number of bits available for domain separation across signatures.

Fortunately, while pairwise collisions may occur, larger multicollisions remain unlikely and their probability can be quantified. We report in Table 22 the minimum number of signature queries Q_{sig} for which the probability of obtaining a $(\mu+1)$-collision in the domain-separation values used as cipher keys exceeds $2^{-\lambda}$, under the random-oracle model. This quantity appears in the EUF-CMA security reductions.

Table 22: Lower bound on the number Q_{sig} of signature queries required for the probability of a $(\mu+1)$-collision in the encryption domain-separation values to exceed $2^{-\lambda}$.

λ	μ	Q_{sig}	
		Correlated-tree	One-tree
128	1	≥ 2	≥ 2
128	2	$\geq 2^{31.1}$	$\geq 2^{25.8}$
128	3	$\geq 2^{51.6}$	$\geq 2^{45.6}$
128	4	$\geq 2^{63.9}$	$\geq 2^{57.5}$
128	5	$\geq 2^{72.1}$	$\geq 2^{65.5}$
192	1	≥ 2	≥ 2
192	2	$\geq 2^{52.5}$	$\geq 2^{47.1}$
192	3	$\geq 2^{83.6}$	$\geq 2^{77.6}$
256	1	≥ 2	≥ 2
256	2	$\geq 2^{73.8}$	$\geq 2^{68.5}$

Table 22 can be interpreted as follows. First, a pairwise collision cannot be considered negligible according to this criterion, since its probability already exceeds $2^{-\lambda}$ with as few as two signature queries. However, higher-order multicollisions require substantially more signatures. In particular, when restricting the number of signature queries to 2^{64}, the probability of a 6-collision remains below 2^{-128} for Category I. Similarly, the probability of a 4-collision remains below 2^{-192} for Category III, while the probability of a 3-collision remains below 2^{-256} for Category V.

Obtaining a salt μ-collision results in a security loss of $\log_2 \mu$ bits. Indeed, if an adversary successfully obtains such a collision, they can perform a multi-target exhaustive search against μ hidden seeds. For instance, such an attack can be easily mounted by targeting the commitments to the hidden leaves of the same parallel repetition across the μ signatures having the salt collision. Seeds along the hidden paths of the GGM trees can similarly be targeted. This security loss is intrinsic to the use of a block cipher with λ-bit keys and λ-bit blocks, as the resulting domain is not large enough to fully separate all cipher calls across up to 2^{64} signatures and all internal calls involving sensitive values. It is therefore shared by all MPC-in-the-Head signature candidates that use a block cipher as their main symmetric primitive, including FAEST [BBB+24] and SDitH [ABB+24].

5.2 Hardness of MQ instances

The security of the MQOM signature scheme relies on the hardness to solve random instances of the Multivariate Quadratic (MQ) problem, since the secret key is a solution of the MQ instance represented by the public key. For the correlated-tree variant, the security relies on the hardness of the (weaker) PGOW-MQ problem, which is addressed in Section 5.3.

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. In particular, the best algorithms are quite different over $\mathbb{F}_2$ and over larger finite fields.

5.2.1 Mathematical tools and building blocks

Arithmetic over $\mathbb{F}_{16}$. We consider that addition over $\mathbb{F}_{16}$ costs 4 bit operations. Two degree-3 polynomials over $\mathbb{F}_2[X]$ can be multiplied in 25 bit operations by schoolbook multiplication. Once their degree-6 product has been computed, it must be reduced modulo the irreducible polynomial that defines the finite field. Take $F = X^4 + X + 1$ (as given in Table 6). The remainder of a degree-6 polynomial modulo F can be computed with 6 bit operations. Therefore we consider that multiplication requires 31 bit operations.

Monomials. Over an arbitrary finite field $\mathbb{F}_q$, we work in the quotient algebra

$$\mathcal{R}_q = \mathbb{F}_q[x_1, \ldots, x_n] \,/\, \langle x_1^q - x_1, \ldots, x_n^q - x_n \rangle \,,$$

so we take into account the so-called "field equations". In particular, each variable appears with exponent at most $q-1$ in a monomial. The number of degree-d monomials of $\mathcal{R}_q$ is the degree-d coefficient of the series

$$\left(\frac{1-z^q}{1-z}\right)^n = \left(1 + z + \cdots + z^{q-1}\right)^n \,,$$

while the number of monomials of degree at most d is the degree-d coefficient of the series

$$\frac{1}{1-z}\left(\frac{1-z^q}{1-z}\right)^n \,.$$

Note that the number of degree-d monomials is precisely $\binom{n+d-1}{d}$ when $d < q$. We denote by $\#\mathcal{M}_q(n,d)$ and $\#\mathcal{M}_q(n,\leq d)$ the number of monomials of degree exactly d and at most d, respectively, in $\mathcal{R}_q$.

Macaulay matrices. Consider a sequence of quadratic polynomials $f_1, \ldots, f_m$ in $\mathcal{R}_q$. Denote by I the ideal they span. I can be seen as an infinite-dimensional vector space spanned by the $\mathfrak{m}f_j$, where $\mathfrak{m}$ ranges across all possible monomials of $\mathcal{R}_q$. Let I_d (resp. $I_{\leq d}$) denote the subspace formed by the $\mathfrak{m}f_j$ where $\mathfrak{m}$ ranges across all monomials of degree $d-2$ (resp. at most $d-2$). In general, I_d is not equal to the set of all degree-d polynomials of the ideal 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 nevertheless equal when the f_i's are homogeneous.

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 $\mathfrak{m}f_j$ with $\deg \mathfrak{m} \leq d-2$ and whose columns correspond to all possible monomials of $\mathcal{R}_q$ of degree at most d. It follows that the row span of the degree-d Macaulay matrix is exactly $I_{\leq d}$. The matrix has $\#\mathcal{M}(n, \leq d)$ columns and $m \cdot \#\mathcal{M}(n, \leq d-2)$ rows.

It is quite sparse, because each row has at most $(n+1)(n+2)/2$ non-zero coefficients. It is well-known that Macaulay matrices are rank-deficient: there are linear dependencies between rows, at least because of the trivial relations $f_i f_j = f_j f_i$. Under the assumption that the f_j form a (semi-)regular sequence, then the above "trivial relations" between the f_i's are the only ones. The 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. This assumption is particularly mild in the case of MQOM, where the polynomial systems are purely random. The interested reader is referred to [Bar04; BFS15] for more details. We now assume that the f_j's are (semi-)regular.

Under the assumption of (semi-)regularity, the rank of the degree-d Macaulay matrix can be predicted, and the assumption implies that it only depends on n, m and q, and not on the actual coefficients of the f_j's. Let r_d denote the rank of the degree-d Macaulay matrix of $f_1, \ldots, f_m$. Then the sequence given by $a_d = \#\mathcal{M}_q(n, \leq d) - r_d$ coincide with the coefficients of the series expansion given in [BCT+25]:

$$\frac{(1-x^q)^{n+1}}{(1-x)^{n+1}} \frac{\left(1-x^2\right)^m}{(1-x^{2q})^m},$$

as long are these are positive.

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 Compute the "Krylov sequence" $(uA^i v)_{i \geq 0}$, where u and v are fixed matrices of respective dimensions $\tilde{m} \times N$ and $N \times \tilde{n}$. In fact, $\tilde{n}$ independent sequences are computed by splitting v in $\tilde{n}$ vectors (this has the advantage of providing coarse-grain parallelism). Each of these $\tilde{n}$ sequences does approximately $(1/\tilde{n} + 1/\tilde{m})N$ iterations. Each iteration does a "sparse matrix-dense vector" product as described above, followed by a (dense $\tilde{m} \times N$ matrix) $\times$ (dense 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 sequences. 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 Compute the linear recurrence relation of the Krylov sequence: a sequence $F_0, F_1, \ldots, F_{D-1}$ of $\tilde{n} \times \tilde{n}$ matrices such that

$$\sum_{i=0}^{D} \left(uA^{k+i}v\right) F_i = 0 \qquad \text{(for all } k \geq \text{some small } k_0\text{)}.$$

Its input consists of the $(\tilde{m} + \tilde{n})N$ field elements stored persistently in BW1. It has quasi-linear running time complexity, and parallelizes well. Its memory usage may not be negligible though. Its actual running time is difficult to predict at this stage.

BW3 Assemble the solution from the linear recurrence. This phase can be split in a high number of independent jobs (if appropriate checkpoints have been taken in BW1). It does at most $N/\tilde{n}$ iterations, where an iteration contains a "sparse matrix-dense vector" plus a (dense $N \times \tilde{n}$ matrix)-vector product. This requires $N(|A|/\tilde{n} + N)$ arithmetic operations in total.

5.2.2 Solving polynomial systems over $\mathbb{F}_{16}$

The XL-Wiedemann 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 implementation works in particular over $\mathbb{F}_{16}$.

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 the block Wiedemann algorithm in [CCN+12] to use the trick described in [CN26] where 1) the matrix u is essentially a projection, so multiplication by u is free, and 2) the BW3 step is essentially free.

The number of columns of A is the number of monomials of degree at most d in n variables in $\mathcal{R}_q$. 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 $N := \#\mathcal{M}_q(n, \leq d)$ 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:

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

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 $\tilde{n}/\tilde{m} \approx 0$. This gives a lower-bound on the number of operations.

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 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 solving degree. The resulting Macaulay matrices are thus much smaller.

In addition, the XL-Wiedemann algorithm only works if the input polynomial span a maximal ideal, and this happens with reasonably high probability as soon as $m \geq n+2$. This implies that at least two variables must be guessed.

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 complexity of the XL-Wiedemann algorithm.

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 large precomputation to accelerate the subsequent resolution of polynomial systems after variables have been "guessed".

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 polynomial systems 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's 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 coefficients of the matrix after the end of the preprocessing. 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.

Recent work. While preparing this third-round update, we were given access to a draft article by Anthropic [CS26] that revisits the concrete cost of solving the MQ instances underlying an MQOM v2 public key. The proposed technique applies the Kronecker solver [GLS01] as extended to arbitrary finite fields by Cafure and Matera [CM06], together with an explicit bit-operation count for the resulting circuit. In the conservative computational model used throughout our specification — where bit operations are counted but memory access is not priced — this yields a modest improvement over our own estimates, most notably for the $\mathbb{F}_{256}$ parameter sets (which we had already excluded from this third-round update). Over $\mathbb{F}_{16}$, this algorithm has complexity $\approx n^{4.5}2^{2n}$, whereas the WXL algorithm has asymptotic complexity $\approx n^2 2^{2.05n}$, with both complexities being of comparable orders of magnitude. Specifically, for the $\mathbb{F}_{16}$ instances of MQOM v2, the article reports bit complexities of $2^{141.8}$ (Category I), $2^{199.9}$ (Category III) and $2^{265.6}$ (Category V), respectively 1.2, 7.1 and 6.4 bits below the corresponding NIST target levels (2^{143}, 2^{207}, 2^{272}). We stress that these complexities are obtained in the conservative computational model where memory access is free and while the underlying attack requires a large amount of memory. Considering other memory cost models, the obtained complexities are higher than the NIST target levels. We are still in the process of independently verifying these results and will update our specification accordingly once this analysis is complete.

Parameters, estimations and discussion. Table 23 shows our choice of parameters for $q = 16$, along with our estimations of the complexity of running the (hybrid-)XL-Wiedemann (WXL) algorithm and the "Polynomial XL" (PXL) algorithm. We decided to slightly increase these parameters compared to the MQOM v2 instances. While the results of [CS26] are still to be verified, this choice offers some security margin — even should these results be confirmed — and is further motivated by the fact that this increase has a negligible impact on timing performance and only a very moderate impact on the signature size and the public key size. Morever, selecting $m = n$ as a multiple of 32 is implementation-friendly, as it allows a full usage of vector registers on modern CPUs.

5.2.3 Solving Boolean polynomial systems

Estimating the difficulty of solving Boolean polynomial system is challenging because of the tension between impractical 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

	Level	I	III	V
	n	64	96	128
WXL	k	15	17	25
	d	14	23	27
	$\log_2 N$	45.1	75.1	92.3
	$\log_2 \lvert A\rvert$	55.4	86.8	104.7
	cost	165.6	235.1	302.1
PXL	k	14	21	30
	d	15	20	24
	$\log_2 \alpha$	35.9	53.0	67.1
	$\log_2 \lvert A\rvert$	88.1	126.6	159.3
	cost	161.7	237.7	313.3

Table 23: Hardness estimates for the $q = 16$, and parameters of the relevant algorithms. k denotes the number of guessed variables in the hybrid method, and d the degree of the macaulay matrix that is built. For WXL, N denotes the size of the (sparse) Macaulay 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 total number of field coefficients. In both cases, $|A|$ is thus a reasonable estimate of the size of the matrix.

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 records. 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 [Din21b], 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 cryptology community with complexity $\mathcal{O}\left(n^2 2^{0.815n}\right)$ using less than $n^2 2^{0.63n}$ 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 assess the efficiency of the Crossbred algorithm [JV17]. 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 $x = (y_1, \ldots, y_{n-k}, z_1, \ldots, z_k)$. 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 degree-D Macaulay matrix in which some columns have been removed. When $d = 1$, which is the most sensible choice, 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. This result in a linear system $Az = b$ where the coefficients of A are degree-$(D-1)$ polynomials in the y variables, and those of b are degree-D polynomials in the y variables. Solving the original quadratic system is done by trying all $y \in \{0,1\}^{n-k}$, evaluating the polynomials in A and b, and finally solving the resulting $k \times k$ linear system in the z variables. In short, instead of doing an exhaustive search on n bits, it solves 2^{n-k} linear systems in k variables. Increasing k is better, but this requires increasing the degree D of the Macaulay matrix (not all combinations of D, k are admissible). The running time of the Crossbred algorithm is the cost of running its preprocessing (which is not negligible) plus the running time of this search phase. Evaluating the polynomials in A and b can be done with the FES algorithm, leading to an estimated cost for the search phase of:

$$2^{n-k}\left(k^2(D-1) + kD + k^3\right).$$

We have significant 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 $\mathbb{F}_2$.

A reduced-space Crossbred variant. 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 $(Sx) = x_1x_2 + x_3x_4 + \cdots + x_{n-1}x_n +$ (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 is shown in [BS26] that, given *two* Boolean quadratic polynomials f and g, there is (often) a linear change of variables S such that $f(Sx)$ and $g(Sx)$ become linear functions after half the variables have been guessed. 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. 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 [Din21a]. 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 $d2^n$, using only twice the amount of memory needed to store the polynomial.

Dinur's "1.9-th algorithm". Bouillaguet and Sauvage give in [BS26] a slightly modified version of Dinur's second algorithm that runs a bit faster. Because it is a hybrid between Dinur's second algorithm and an earlier algorithm of [LPT+17], they dubbed it "Dinur's 1.9-th algorithm". It is structurally similar to Dinur's second algorithm but make different compromises, leading to a small decrease in complexity (a few bits at most).

Dinur's first algorithm. While having the best asymptotic complexity, Dinur's first algorithm [Din21b] 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 believe that, because the algorithm is complex and has never been implemented, any concrete estimation of its complexity should be taken with caution.

Parameters, estimations and discussion. Table 24 gives our estimates of the cost of running relevant Boolean solving algorithms. Over $\mathbb{F}_2$, we deliberately take a more aggressive stance compared to $\mathbb{F}_{16}$, with the objective to keep the signature size as low as possible – in particular below 2.5 KB for Category I. For this reason, our choice of parameters is based on the following considerations.

Cost of memory accesses. There is no consensus in the cryptologic community regarding the cost of accessing a large memory, which may further depend on the actual access pattern of

the algorithm. We consider that this debate cannot be settled by experiments at the present time. Our opinion is that it is difficult to say anything sensible about the "practical cost" (on machines that do not exist) of a computation whose parameters were chosen precisely to make it impossible. Therefore, we make the conservative choice of not "charging" memory accesses.

Large memory requirements. The Earth is composed of about 2^{165} atoms (a large majority of which is inaccessible), therefore attacks with space requirements close to this number are deemed unrealistic.

Implausibility of large-space attacks. Even below the 2^{165}-bit threshold, we argue that attacks with large space requirements are not realistic threats. As pointed out by Bernstein [Ber05], assuming that a machine large enough to hold an extremely large amount of memory (e.g. capable of holding 2^{100} bits) could be built seems akin to assuming that a parallel machine large enough to break the AES by exhaustive search in a reasonable amount of time could also be built: simply replace the hardware that stores bits by hardware that evaluates the AES. The "convertion ratio" between space and computation is technology-dependent, so we can only provide a very crude estimate that is only meaningful at the time of writing. We only consider chip surface and neglect everything else (energy, interconnect, thermal dissipation, ...). [PLK+25] describes a 3D NAND Flash chip storing 28Gb/mm^2, while [MSK+11] describes a circuit occupying 0.052mm^2 using a 45nm CMOS process that does $2^{28.6}$ AES-128 encryptions per second with a fixed key. Adding key expansion increases area by $\approx 30\%$. Scaling down from 45nm to 3nm multiplies density by ≈ 50 and frequency by ≈ 2, leading to $\approx 2^{39}$ encryptions per second per mm^2. So, 1mm^2 of machine surface can be used to store $\approx 2^{35}$ bits or perform $\approx 2^{39}$ AES evaluations per second. Under this estimate, an attack that requires 2^{100} bits of storage requires a (very theoretical) machine as large as another (very theoretical) machine that breaks the AES-128 by exhaustive search in 5 months. Therefore, taking into consideration the "danger" posed by an attack that requires 2^{100} bits of space implicitly assumes that the AES-128 is already practically vulnerable.

Simplified non-uniform computational model. Some algorithms are analyzed in a Boolean circuit model represented by straight-line programs:

> *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).* [Din21a]

In other terms, a statement such as:

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

counts as a single bit operation. Clearly, just *producing* the straight-line program in the first place requires more work than simply *executing* it. Switching to a computational model where the adversary executes a compact program written in a usual programming language would likely drive the computational cost up by an unknown factor, because the adversary would then have to generate the circuit "on the fly".

This is in particular the case of the memory-efficient Möbius transform, one of the dominating steps in Dinur's algorithms. Bouillaguet [Bou24] has shown that this operation 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 than $d2^n$ bit operations, as claimed in [Din21a]. Experiments in [Bou24]

suggest at least 20 real-world CPU cycles per bit operation in the simplified model. Therefore, we consider a more realistic model in which we assume that the cost of the memory efficient Möbius transform is about $20nd2^n$ "classical gates" (this assumes that an elementary operation on n-bit words translates to exactly n gates). This is slightly pessimistic for the adversary: it is plausible be that the constant 20 could be lowered by an improved implementation. But this will not affect the outcome of our estimates.

For the sake of completeness, we also give cost estimates for algorithms dominated by the memory-efficient Möbius transform in the simplified model where the memory-efficient Möbius transform requires $d2^n$ classical gates. These are shown in Table 25.

Conclusion. For all these reasons, we consider that Dinur's algorithms are not realistic threats to the proposed MQOM parameters, even if the number of classical gates in the simplified unrealistic computational model with impossibly large memories drops 1 or 2 bits below the expected security threshold. We consider that the Crossbred algorithm (especially its low-space variant that we dub Crossbred$^+$) is the most "dangerous" threat at the time of writing this specification.

Real-world estimate. 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.

5.3 Hardness of PGOW-MQ

As formally introduced in Definition 2, the PGOW-MQ problem is the following:

- The adversary receives a quadratic system composed of m equations in n variables over $\mathbb{F}_q$. Assume that λ is a multiple of $\log_2 q$ and set $k := \lambda / \log_2 q$.
- The adversary has access to an oracle. The adversary may submit candidates $c \in \mathbb{F}_q^k$ to the oracle. The oracle returns $\top$ if the submitted candidate is a prefix of a complete solution of the system, and $\bot$ otherwise.
- The adversary wins if they eventually return such a prefix.
- The adversary (T, q, ϵ)-solves the PQOW-MQ problem if it runs in time T, submits at most q queries to the oracle and returns a partial solution of the system with probability at least ϵ.

In this section, we argue that this problem is hard.

5.3.1 Hardness of finding a solution prefix

Suppose that an algorithm $\mathcal{A}$ recovers the first k coefficients of a solution of a system of n quadratic equations in n variables over $\mathbb{F}_q$. To recover the complete solution, proceed as follows:

1. Run $\mathcal{A}$ to obtain a prefix of a solution.
2. Substitute the value of these variables into the original polynomial system.
3. Solve the resulting system of m equations in $n - k$ variables using e.g. the Crossbred algorithm (if $q = 2$) or the hybrid-XL-Wiedemann algorithm (if $q = 16$).

	Level	I	III	V
	n	160	240	320
Crossbred				
	h	3	4	3
	D	13	20	27
	k	31	44	56
	Space	75	111	145
	Cost	144	213	282
Crossbred$^+$				
	h	86	122	160
	D	5	8	11
	k	26	38	51
	Space	37	54	70
	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	112	164	215
	Cost	150	216	283
Dinur 1.9th				
	k	33	50	65
	ℓ	36	52	68
	τ	5	13	12
	σ	5	11	11
	Space	112	164	218
	Cost	147	212	278

Table 24: Hardness estimates of the Boolean parameter sets, in the more realistic computational model where the memory-efficient Möbius transform requires $20nd2^n$ classical gates.

Because this solves the original system, we assume that the total running time of this procedure cannot be less than the estimates given in Table 23 and Table 24 (otherwise our estimates

	Level	I	III	V
	n	160	240	320
Dinur 1st				
	n_1	37	58	80
	n_2	31, 23, 17, 0	52, 41, 33, 26, 20, 0	74, 61, 51, 43, 36, 30, 24, 18, 0
	Space	132	191	249
	Cost	151	211	270
Dinur 2nd				
	n_1	30	45	60
	Space	108	159	210
	Cost	142	208	274
Dinur 1.9th				
	k	29	47	62
	ℓ	31	49	64
	τ	16	18	28
	σ	13	15	23
	Space	108	162	213
	Cost	141	205	270

Table 25: Hardness estimates of the Boolean parameter sets, in the unrealistic simplified computational model where the memory-efficient Möbius transform requires $d2^n$ classical gates.

would have been incorrect in the first place). We claim that the third step has low complexity and that it is practical in most cases. This implies that the complexity of the first step is comparable to that of solving the original system.

When $q = 2$, the third step of this procedure requires solving a Boolean system of m equations in n unknowns, where (n, m) is $(32, 160), (48, 240)$ or $(64, 320)$ for security levels I, III and V, respectively. Random systems of these sizes are practically easy to solve: the `hpXbred` software solves them in 0.1s, 1s and 750s, respectively, using a single core on a laptop. The estimated workload is $2^{37}, 2^{40}$ and 2^{53} bit operations, respectively.

When $q = 16$, the third step of the above procedure requires solving a system of m equations over $\mathbb{F}_{16}$ in n unknowns, where (n, m) is $(32, 64), (48, 96)$ or $(64, 128)$ for security levels I, III and V, respectively. The computational cost of these three subproblems is estimated at 2^{60}, 2^{79} and 2^{95} respectively.

We conclude that, for our considered instances, recovering the first λ bits (k variables) from the solution of a system of n quadratic equations in n variables over $\mathbb{F}_q$ is as hard as completely finding the solution of the system.

5.3.2 Reduction to standard MQ

In this section, we give a simple but lossy reduction from PGOW-MQ to MQ. Given our estimates on the hardness of MQ, this automatically gives (weaker) estimates on the hardness

of PGOW-MQ.

Suppose that $\mathcal{A}$ is an algorithm that (T, Q, ϵ)-solves PGOW-MQ. In order to turn $\mathcal{A}$ into an algorithm that solves MQ, we need to be able to answer its oracle queries. When $\mathcal{A}$ sends a candidate prefix to the oracle, we to simulate the oracle and check whether this prefix extends to a complete solution of the input system. This incurs a cost, detailed above, depending on (n, m, q), that we denote by $T_{MQ}(n-k, m, q)$. This yields an algorithm that $(T + Q \cdot T_{MQ}(n-k, m, q), \epsilon)$-solves MQ. Our estimates on the hardness of MQ yields lower-bounds on $T + Q \cdot T_{MQ}(n-k, m, q)$.

In the context of MQOM, an adversary that has access to a single signature can answer oracle queries with one evaluation of the AES. Therefore, an algorithm that (T, Q, ϵ)-solves PGOW-MQ yields an attack that $(T + Q \cdot T_{AES}, \epsilon)$-breaks MQOM, where T_{AES} is 2^{15} or 2^{16}, depending on the security level. Therefore, the running time of the attack is lower-bounded by $T_{MQ}(n, m, q) \cdot T_{AES}/T_{MQ}(n-k, m, q)$. When instantiated with the concrete parameters, this yields concrete lower-bounds on the cost of an attack against MQOM, shown in Table 26. The reduction is not tight and entails a security loss of 20 to 65 bits, depending on the parameter set.

q	Level	n	Attack lower-bound
2	I	160	122
	III	240	188
	V	320	245
16	I	64	121
	III	96	171
	V	128	223

Table 26: Lower bound on the security of PGOW-MQ by reduction to MQ.

5.3.3 Cryptanalysis of PGOW-MQ

Compared to an adversary trying to solve an instance of MQ, an adversary against the same instance of PGOW-MQ has a weaker goal and extra help: they only need to find a prefix of the solution and they may use the oracle to test guesses. In this section, we argue that PGOW-MQ is, for all practical purposes, likely to be just as hard as MQ itself.

The main point of our reasoning is the following: in most cases, what the oracles does, the adversary could do by themselves. For instance, the oracle is only useful if the adversary finds a candidate *partial* solution. Indeed, if the adversary has a candidate *full* solution, they could just evaluate all the polynomials and check if they all vanish, without using the oracle at all. Also, given a candidate partial solution, the adversary could try to obtain a full solution by solving the resulting overdetermined instance of MQ, as outlined in the reduction above. Arguably, this takes more time than invoking the oracle. So, if the adversary is capable of isolating a *small* number of candidate partial solution, then they can already afford this extra cost. Only if the adversary produces a *large* list of candidate partial solutions is the oracle useful to sieve them.

In any case, finding candidate partial solutions better than by random chance is non-trivial, and any efficient way to do it would almost surely yield better algorithms to solve polynomial systems.

Polynomial elimination. An idea that naturally comes to mind in this context is polynomial elimination: searching inside the ideal I spanned by the original equation for polynomials in which only the first k variables appear (this corresponds to the size of prefixes received by the oracle). These polynomials belong to the k-th elimination ideal of I, that is $I \cap \mathbb{F}_q[x_1, \ldots, x_k]$. They admit as solutions the prefixes of the original solutions, plus potentially some parasitic solutions. However, the set of parasitic solutions will be very small, and even most likely empty (this statement can be made precise by the Closure theorem, see [CLO91, ch.3, §2]).

If the adversary could find many "random" polynomials in the k-th elimination ideal, then they could assemble a polynomial system whose solutions are the partial solutions to the input system (plus a few parasitic solutions). This would enables the adversary to test candidate partial solutions without using the oracle, with a high degree of accuracy, and would make the oracle redundant.

If, on the other hand, the adversary could only find a few polynomials in the k-th elimination ideal, then the resulting polynomial system would have many solutions and the oracle could potentially be useful to test them.

We now argue neither of these two scenario can actually happen, because finding polynomials in the k-th elimination ideal is too hard.

A degree-d polynomial has $\#\mathcal{M}_q(n, \leq d)$ monomials, of which $\#\mathcal{M}_q(n-\ell, \leq d)$ do not include the last ℓ variables. Given $\#\mathcal{M}_q(n, \leq d) - \#\mathcal{M}_q(n-\ell, \leq d)$ linearly independent degree-d polynomials in the ideal I, we can perform a linear combination in which the last ℓ variables do not appear. For this to happen, the rank of the degree-d Macaulay matrix must be large enough to provide sufficiently many linearly independent degree-d polynomials, and this imposes a lower-bound on d (typically an increasing function of ℓ). If d is large enough for such polynomials to exist, then they can be found using the block-Wiedemann algorithm applied to truncated Macaulay matrices (finding left-kernel vectors in the Macaulay matrix where the columns corresponding to monomials containing only the first $n-\ell$ variables have been removed). This enables an estimation of the required number of operations. Table 27 shows such estimates.

Finding polynomials in which only the first k variables (λ bits) of the solution appear is too costly (it always costs more than solving the system). So the scenario where the adversary can find random polynomials in the k-th elimination ideal and "build the oracle themselves" is excluded.

However, it is possible to find polynomials in which a smaller number of variables have been eliminated. This can even be practical if the number of eliminated variables is small enough (for instance, eliminating just one variable is often doable with polynomials of degree 3 which can be easily computed). Table 27 shows the maximum number of variables that can be eliminated faster than actually solving the system.

When it is feasible, finding several random polynomials is almost at easy as finding just one: running the block Wiedemann algorithm will produce several random ones, and the procedure can be repeated a few times.

In the end, this approach does not really seem to make much progress: the adversary is still facing an instance of PGOW-MQ, but instead of n quadratic polynomials in n variables, they now have several large-degree polynomials in $n-\ell$ variables, along with an oracle for partial solutions of size k (and $k < n-\ell$ so there are still "too many variables").

The same procedure could be attempted to solve the input system: find higher degree polynomials in the first $n-\ell$ variables, solve the resulting higher-degree system to obtain partial solutions, and finally extend these into solutions of the original system. This approach is never successful.

			Full elimination			Partial elimination		
q	Level	n	ℓ	degree	cost	ℓ	degree	cost
2	I	160	32	19	177	15	13	145.5
	III	240	48	27	253	23	19	208.1
	V	320	64	35	329	34	26	277.3
16	I	64	32	36	198	17	19	139.1
	III	96	48	51	284	27	29	206.2
	V	128	64	65	366	36	38	268.9

Table 27: Cost for finding polynomials in which the last ℓ variables do not appear. "Full elimination" corresponds to the case where only the first λ bits of the solution appear in the result, whereas "Partial elimination" consists in eliminating as many variable as possible while staying "in-budget".

Comparison with the "polynomial method". We note that, over $\mathbb{F}_2$, algorithms that are incarnations of the "polynomial method" [LPT+17; BKW19; Din21b; Din21a] actually invest nearly all their running time into building a mechanism that enables them to test partial solutions efficiently. So, they implement the functionality offered by the PGOW-MQ oracle themselves. More precisely, they build (approximations of) the large-degree polynomial in $n - \ell$ variables that evaluates to 1 when its input is a partial solution and 0 otherwise. Once it is built, these algorithms evaluate it on all possible inputs to find partial solutions. Because they can choose ℓ freely, they can balance the cost of the two phases (building the polynomial *vs.* evaluating it).

Building on this observation, Bouillaguet and Sauvage [BS26] designed a variant of Dinur's second algorithm dedicated to solving Boolean PGOW-MQ. The algorithm works by investing a significant amount of time in building an approximate oracle for partial solutions that fails to reveal an actual partial solution with probability less than 1/2 while reporting spurious partial solutions with a probability that can be controlled. After that, each suggested partial solution is submitted to the oracle.

In the context of MQOM when an oracle invocation can be implemented by evaluating the AES, this yields an attack that is marginally faster than directly attacking the MQ instance exposed by the public key, in the Boolean setting. The procedure described in [BS26] also relies on the memory-efficient Möbius transform, so the caveats discussed above also apply. In the more realistic setting where the memory-efficient Möbius transform costs $20nd2^n$ bit operations, the algorithm degenerates into exhaustively submitting all potential prefixes to the oracle. We show estimates in the simplified unrealistic setting where the memory-efficient Möbius transform costs $d2^n$ bit operations in Table 28.

This suggests that, in the Boolean setting, solving the PGOW-MQ instance exposed by a single signature is marginally easier (1 to 2 bits) than trying to solve the quadratic system exposed by the public key, in the simplified unrealistic computational model.

Conclusion. In conclusion, over $\mathbb{F}_{16}$ there does not seem to be any way to solve PGOW-MQ faster than MQ itself. In the Boolean setting, the situation is slightly different and the numbers given in Table 28 again drop 3 to 4 bits below the expected security levels. However, we do not believe that these numbers are meaningful, for the same reasons given above: the obscenely large memory consumption of these algorithms and the fact that they can only become better than an exhaustive search on the AES in a simplified and unrealistic computational model.

n	160	240	320
λ	128	192	256
$\log_2 T_{AES}$	15	15	16
ℓ	33	49	65
τ	8	8	8
σ	6	6	6
Space	106.0	158.5	210.3
Cost	140.0	203.0	268.5

Table 28: Estimates of number of bit operations required to break MQOM by solving a PGOW-MQ using the algorithm of [BS26], in the simplified unrealistic computational model.

We therefore believe that for our proposed parameters over $\mathbb{F}_2$ and $\mathbb{F}_{16}$, the PGOW-MQ problem is as hard as MQ itself, and that the security of MQOM is therefore not significantly affected by the existence of the oracle.

6 Design choices

This section presents the design rationale of MQOM v3 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.

6.1 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 since 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 29, we present the signature sizes for a variant of MQOM v3 based on the VOLEitH framework. We apply the same optimizations to this variant as in the TCitH-based version, including tree optimizations and grinding (see the following subsections). Our results show that the short and shorter signature variants yield comparable sizes across both frameworks, while the fast signature variants are slightly smaller with the VOLEitH framework (but short sizes is not the primary goal for the trade-off fast which rather targets efficient running times). 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.

Independent batching challenges. In the considered PIOP protocol (see Section 2.1.2), the first verifier challenge is used to batch multiple polynomials in order to reduce their communication cost. Instead of sending the $\hat{m}$ coordinates of the vector polynomial P_z (masked by P_u), the prover sends only $\eta < \hat{m}$ random linear combinations of them, namely the vector polynomial $\Gamma \cdot P_z$ (still masked by P_u, yielding the vector polynomial P_α).

In MQOM v2, two variants were proposed for each instance: one relying on batching (five-round variant) and one without batching (three-round variant). The variant without batching

Framework	MQOM3 – TCitH	VOLEitH	Difference
MQOM3-L1-gf2-shorter-ct	2 492 B	2 554 B	+2%
MQOM3-L1-gf2-shorter-ot	2 492 B	2 522 B	+1%
MQOM3-L1-gf16-short-ct	2 932 B	2 890 B	−1%
MQOM3-L1-gf16-short-ot	2 932 B	2 890 B	−1%
MQOM3-L1-gf16-fast-ct	3 316 B	3 202 B	−3%
MQOM3-L1-gf16-fast-ot	3 316 B	3 202 B	−3%
MQOM3-L3-gf2-shorter-ct	5 932 B	5 956 B	+0%
MQOM3-L3-gf2-shorter-ot	5 890 B	5 956 B	+1%
MQOM3-L3-gf16-short-ct	6 556 B	6 638 B	+1%
MQOM3-L3-gf16-short-ot	6 556 B	6 446 B	−2%
MQOM3-L3-gf16-fast-ct	7 564 B	7 298 B	−4%
MQOM3-L3-gf16-fast-ot	7 660 B	7 204 B	−6%
MQOM3-L5-gf2-shorter-ct	10 836 B	10 540 B	−3%
MQOM3-L5-gf2-shorter-ot	10 804 B	10 454 B	−3%
MQOM3-L5-gf16-short-ct	12 100 B	11 408 B	−6%
MQOM3-L5-gf16-short-ot	11 716 B	11 410 B	−3%
MQOM3-L5-gf16-fast-ct	13 540 B	13 058 B	−4%
MQOM3-L5-gf16-fast-ot	13 380 B	12 804 B	−4%

Table 29: Comparison of MQOM v3 with its variant over VOLEitH. All the signature size are in bytes. While τ would be larger using TCitH than using VOLEitH, the number of the expanded bits using PRG on tree leaves is larger using VOLEitH than using TCitH, because of the mask for the consistency check.

incurred a modest increase in signature size but benefited from a simpler design, as the underlying zero-knowledge proof was a three-round protocol (a.k.a. Sigma protocol). Moreover, this three-round variant was easier to implement and offered practical implementation advantages. In particular, because no batching was involved, it was possible to compute the BLC commitment and execute the PIOP protocol for a given parallel repetition independently of the others. By contrast, in the batched variant, the batching challenges could only be computed after all BLC commitments had been generated, preventing the complete execution of one repetition before all commitments were available. This independence is particularly useful for designing memory-efficient or streaming implementations.

In MQOM v3, we remove the three-round variants and retain only the size-optimal five-round variants (using batching). However, to preserve the implementation advantages of the three-round variant, we slightly modify the batching procedure. Instead of deriving all batching challenges from a single hash of the complete set of commitments, we derive the batching challenge of each parallel repetition by hashing only the commitment corresponding to that repetition. This restores the independence between parallel repetitions and therefore recovers all the implementation advantages of the three-round variant (except for its simpler design), while retaining the communication benefits of batching. This design choice is *sound* because the batching soundness error is already *negligible for each individual repetition.*

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, 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 since MQOM v2.

6.2 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 $y = x \oplus z$.

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 $\mathsf{seed} \parallel \mathsf{PRG}(\mathsf{seed})$, 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.

However, [FR26; KX26] show that the security of schemes based on correlated trees cannot be reduced directly to the hardness of the underlying computational problem. More precisely, these works show that security additionally relies on the hardness of recovering the first λ bits of the extended witness given access to a validation oracle, which determines whether a candidate string matches the first λ bits of the witness. The computational cost of each oracle query is essentially that of a single AES evaluation. The difficulty of this auxiliary problem depends heavily on the structure of the witness. In the case of MQOM, it amounts to recovering the first $\lambda/\log_2|\mathbb{F}|$ coordinates of the MQ solution using access to this validation oracle.

One-tree optimization [BBM+24]. Some combinatorial optimizations of the GGM trees have 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 $(\tau \cdot i + e)^{\text{th}}$ 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 revealed nodes in the revealed sibling paths exceeds a threshold, the signer rejects the challenge and recomputes 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 a secure scheme even if the challenge space is reduced, because the security loss is compensated by the computational cost of searching a valid challenge. This optimization is not compatible with the correlated-tree optimization.

Relaxed vector commitment [KLS24]. An additional 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.

Design choices. For MQOM v3, we decided to support two variants:

- A variant relying on the *correlated-tree optimization*. The security of this variant cannot be tightly reduced to the hardness of the MQ problem itself, but instead relies on the hardness of the weaker PGOW-MQ problem, in which the adversary is additionally given access to a validation oracle for the first λ bits of the MQ solution. Nevertheless, there

is no known reason to expect this additional oracle access to make the problem significantly easier in practice (see Section 5.2). On the other hand, this variant is significantly easier to implement and offers several practical advantages, including better parallelism, streamability, and memory efficiency. As a result, it appears particularly well suited for deployment on a wide range of platforms.

- A variant relying on the *one-tree optimization*. This variant is intended to provide a conservative alternative with a tight reduction to the MQ problem. However, the resulting implementations are less implementation-friendly. While they achieve good overall performance on desktop-class platforms, they may be less suitable for more constrained environments, such as memory-limited devices or hardware implementations.

Supporting both variants is also motivated by the desire to provide a fair comparison between these two optimization techniques. Since they are integrated into the same framework, they can be evaluated under identical assumptions and implementation choices.

A third possibility would be to avoid both optimizations altogether (except possibly the half-tree optimization). This would simply consist of using OT.SeedExpand instead of CT.SeedExpand in the pseudo-code of the correlated-tree variant, while modifying the BLC construction so that Hash_7 takes the entire vector $\Delta_x[e]$ as input. Such a variant would retain the implementation advantages of the correlated-tree construction while enjoying the same tight reduction to the MQ problem as the one-tree variant. The drawback of this non-optimized variant is an increase in signature size (up to 10%). Table 30 reports the signature sizes of this non-optimized variant. Its computational performance should be essentially identical to that of the correlated-tree variant.

Finally, we did not include relaxed vector commitments in MQOM v3, as it is currently unclear whether they can be combined soundly with the TCitH framework.

Table 30: Signature sizes of the non-optimized variant.

Parameter Set	**Sizes (in bytes)**		
	Correlated-tree	One-tree	Non-optimized
MQOM3-L1-gf2-shorter	2 492	2 492	2 652 (+6%)
MQOM3-L1-gf16-short	2 932	2 932	3 124 (+7%)
MQOM3-L1-gf16-fast	3 316	3 316	3 588 (+8%)
MQOM3-L3-gf2-shorter	5 932	5 890	6 316 (+7%)
MQOM3-L3-gf16-short	6 556	6 556	6 988 (+7%)
MQOM3-L3-gf16-fast	7 564	7 660	8 188 (+8%)
MQOM3-L5-gf2-shorter	10 836	10 804	11 540 (+7%)
MQOM3-L5-gf16-short	12 100	11 716	12 900 (+10%)
MQOM3-L5-gf16-fast	13 540	13 380	14 660 (+10%)

6.3 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, which is known as *grinding* [Sta21]. Together with the query challenge, the Fiat-Shamir hash computation outputs a w-bit value which must be zero for the verification algorithm to accept the signature, where w is a parameter of the scheme. If this value is not zero, the signer rejects the query challenge and recomputes the

hash with an incremented counter, which is repeated 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 have used the grinding optimization since MQOM v2. In that version, the grinding parameter w was chosen to reduce the required number τ of parallel repetitions. In MQOM v3, we replace the hash-based grinding procedure with an AES-based one. More precisely, we adopt the construction proposed by [Riv26]. Besides providing a more efficient grinding procedure, this approach also aligns MQOM more naturally with NIST security Categories I, III, and V, which are defined in terms of the security of the AES block cipher.

6.4 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 since 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, while easing the security analysis in the ideal cipher model.

7 Advantages and limitations

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

- **Relatively slow:** As other MPCitH-based schemes, MQOM is relatively slow, with signing and verification time ranging between 1.13 and 15.4 megacycles (0.43 – 6.24 ms on an AMD Ryzen Threadripper PRO 7995WX processor) for NIST security category I. This is slow compared to lattice-based signatures. One of the reasons 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 from the following advantages:

- **Conservative hardness assumption:** Being generic, the MPCitH approach can be applied to any problem and does not rely on structured problems to introduce a trapdoor. MQOM benefits from this by relying on a fully random instance of the MQ problem which is believed to be a conservative hardness assumption. The correlated-tree variant of MQOM relies on the hardness of the (weaker) PGOW-MQ problem but, as argued in Section 5.3, we believe this problem to be as hard as MQ itself in practice for our choice of parameters.
- **Small (public) keys:** Thanks to the unstructured feature of the MQ instance, it can be mostly derived from a random seed. Hence the public key is only composed of a 2λ-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 could further be fully compressed as a seed).
- **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).
- **Relatively short signatures:** The last generation of MPCitH-based signature schemes in the literature (at time of writing) achieves signature sizes ranging on 2.5–5 KB (for Category I). MQOM is on the lower side of this range, with 2.5–3.3 KB. Among the MPCitH-based NIST candidates selected for Round 3, MQOM has the shortest signatures.
- **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. For our `shorter` variant, this size is of 2.5 KB (for Category I), which is 32% shorter than the 3.7 KB of ML-DSA-44.
- **Fairly embedded friendly:** For an MPCitH-based scheme, MQOM is fairly well-suited for implementation on embedded devices. Benchmarks on Cortex-M4 demonstrate that MQOM can be implemented with low memory requirements (under 10 KB for L1). Among

the MPCitH-based NIST candidates selected for Round 3, MQOM requires the fewest symmetric primitive calls, and natively achieves the lowest memory footprint. Moreover, protecting MQOM against side-channel attacks using masking is simple: it avoids costly arithmetic-Boolean masking conversions (common in lattice-based schemes), does not require any divisions (even with public divisors) or Gaussian elimination (as in UOV-like schemes).

References

[ABB+24] C. Aguilar Melchor, S. Bettaieb, L. Bidoux, T. Feneuil, P. Gaborit, N. Gama, S. Gueron, J. Howe, A. Hülsing, D. Joseph, A. Joux, M. Kulkarni, E. Persichetti, T. H. Randrianarisoa, M. Rivain, and D. Yue. SDitH — Syndrome Decoding in the Head. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures (cited on pages 56, 74).

[AP20] A. Adomnicai and T. Peyrin. Fixslicing aes-like ciphers: new bitsliced aes speed records on arm-cortex m and risc-v. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, (1):402–425, 2020. URL: https://tches.iacr.org/index.php/TCHES/article/view/8739 (cited on page 51).

[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. URL: https://tel.archives-ouvertes.fr/tel-00449609 (cited on page 76).

[BBB+24] C. Baum, L. Braun, W. Beullens, C. Delpech de Saint Guilhem, M. Klooß, C. Majenz, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl. FAEST. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures (cited on page 74).

[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 H. Handschuh and A. Lysyanskaya, editors, *CRYPTO 2023, Part V*, pages 581–615, Santa Barbara, CA, USA. Springer, Cham, Switzerland, 2023 (cited on pages 3–5, 9, 91).

[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 K.-M. Chung and Y. Sasaki, editors, *ASIACRYPT 2024, Part I*, pages 463–493, Kolkata, India. Springer, Singapore, Singapore, 2024 (cited on pages 5, 11, 94, 95).

[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 79).

[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. URL: https://eprint.iacr.org/2024/097 (cited on page 93).

[BCT+25] J. Baena, D. Cabarcas, S. K. Tiwari, J. A. Verbel, and L. Villota. Admissible parameters for the crossbred algorithm and semi-regular sequences over finite fields. *Des. Codes Cryptogr.*, (7):2561–2589, 2025. URL: https://doi.org (cited on page 76).

[Ber05] D. J. Bernstein. Understanding brute force, 2005. URL: http://cr.yp.to/papers.html\#bruteforce (cited on page 83).

[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. URL: https://doi.org/10.1007/978-3-031-15979-4_16 (cited on page 77).

[BF26] R. Benadjila and T. Feneuil. Breaking the myth of MPCitH inefficiency: optimizing MQOM for embedded platforms. Cryptology ePrint Archive, Paper 2026/078, 2026. URL: https://eprint.iacr.org/2026/078 (cited on pages 51, 62, 65).

[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 77).

[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 pages 77, 78).

[BFR24] R. Benadjila, T. Feneuil, and M. Rivain. MQ on my mind: post-quantum signatures from the non-structured multivariate quadratic problem. In *2024 IEEE European Symposium on Security and Privacy*, pages 468–485, Vienna, Austria. IEEE Computer Society Press, 2024 (cited on page 3).

[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 page 76).

[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. URL: https://doi.org/10.1016/j.jco.2012.07.001 (cited on page 79).

[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, Santa Barbara, CA, USA. Springer, Cham, Switzerland, 2020 (cited on page 76).

[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. URL: https://doi.org/10.4230/LIPIcs.ICALP.2019.26 (cited on pages 80, 89).

[Bou24] C. Bouillaguet. Algorithm xxx: evaluating a boolean polynomial on all possible inputs. *ACM Trans. Math. Softw.*, 2024. URL: https://doi.org/10.1145/3699957. Just Accepted (cited on page 83).

[BS26] C. Bouillaguet and J. Sauvage. A decrementally-improved algorithm for boolean mq. Cryptology ePrint Archive, Paper 2026/1704, 2026. URL: https://eprint.iacr.org/2026/1704. https://eprint.iacr.org/2026/1704 (cited on pages 81, 82, 89, 90).

[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 76, 77).

[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, Cambridge, MA, USA. Springer Berlin Heidelberg, Germany, 2005 (cited on page 9).

[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. URL: https://doi.org/10.1007/3-540-45539-6_27 (cited on page 77).

[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 88).

[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 Q. Tang and V. Teague, editors, *PKC 2024, Part I*, pages 229–258, Sydney, NSW, Australia. Springer, Cham, Switzerland, 2024 (cited on page 93).

[CM06] A. Cafure and G. Matera. Fast computation of a rational point of a variety over a finite field. *Mathematics of Computation*, (256):2049–2085, 2006. arXiv:math/0406085 (cited on page 79).

[CN26] T. Chou and R. Niederhagen. A practical optimization for wiedemann XL. Cryptology ePrint Archive, Paper 2026/1787, 2026. URL: https://eprint.iacr.org/2026/1787 (cited on page 77).

[CS26] Claude and Z. Straznickas. A deformation-method solver for square quadratic systems over finite fields, and the large-field parameters of mqom. Draft, Anthropic. Personal communication, 2026 (cited on page 79).

[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. URL: https://doi.org/10.1007/978-3-030-77870-5_12 (cited on page 79).

[Del26] J. L. Delgado. Passive full-key recovery for the MQOM v2 lineage from saltless root expansion. Cryptology ePrint Archive, Paper 2026/1542, 2026. URL: https://eprint.iacr.org/2026/1542 (cited on page 1).

[Din21a] 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, Zagreb, Croatia. Springer, Cham, Switzerland, 2021 (cited on pages 80, 82, 83, 89).

[Din21b] 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. URL: https://doi.org/10.1137/1.9781611976465.151 (cited on pages 80, 82, 89).

[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. URL: https://doi.org/10.1007/978-3-031-62746-0_6 (cited on page 78).

[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 page 3).

[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. URL: https://eprint.iacr.org/2023/1573 (cited on pages 3–5, 9, 10, 91, 93).

[FR23c] T. Feneuil and M. Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In J. Guo and R. Steinfeld, editors, *ASIACRYPT 2023, Part I*, pages 441–473, Guangzhou, China. Springer, Singapore, Singapore, 2023 (cited on page 4).

[FR26] T. Feneuil and M. Rivain. On the security of MPC-in-the-head signatures with correlated GGM trees. Cryptology ePrint Archive, Paper 2026/615, 2026. URL: https://eprint.iacr.org/2026/615 (cited on pages 1, 69, 73, 94).

[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, San Francisco, CA, USA. IEEE Computer Society Press, 2020 (cited on page 15).

[GLS01] M. Giusti, G. Lecerf, and B. Salvy. A Gröbner free alternative for polynomial system solving. *Journal of Complexity*, (1):154–211, 2001 (cited on page 79).

[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, Lyon, France. Springer, Cham, Switzerland, 2023 (cited on pages 10, 93).

[HJ24] J. Huth and A. Joux. MPC in the head using the subfield bilinear collision problem. In L. Reyzin and D. Stebila, editors, *CRYPTO 2024, Part I*, pages 39–70, Santa Barbara, CA, USA. Springer, Cham, Switzerland, 2024 (cited on page 93).

[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, San Diego, CA, USA. ACM Press, 2007 (cited on page 3).

[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 pages 80, 81).

[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, Toronto, ON, Canada. ACM Press, 2018 (cited on pages 3, 9).

[KLS24] S. Kim, B. Lee, and M. Son. Relaxed vector commitment for shorter signatures. Cryptology ePrint Archive, Report 2024/1004, 2024. URL: https://eprint.iacr.org/2024/1004 (cited on pages 93, 94).

[KPR+] M. J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, and K. Stoffelen. PQM4: post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4 (cited on page 50).

[KX26] H. Kosuge and K. Xagawa. Towards formal security proofs of MQOM. Cryptology ePrint Archive, Paper 2026/629, 2026. URL: https://eprint.iacr.org/2026/629 (cited on pages 69, 94).

[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 page 77).

[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. URL: https://doi.org/10.1137/1.9781611974782.143 (cited on pages 80, 82, 89).

[MSK+11] S. Mathew, F. Sheikh, M. E. Kounavis, S. Gueron, A. Agarwal, S. Hsu, H. Kaul, M. A. Anders, and R. Krishnamurthy. 53 Gbps native GF$(2^4)^2$ composite-field aes-encrypt/decrypt accelerator for content-protection in 45 nm high-performance microprocessors. *IEEE J. Solid State Circuits*, (4):767–776, 2011. URL: https://doi.org/10.1109/JSSC.2011.2108131 (cited on page 83).

[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 3).

[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. URL: https://doi.org/10.1007/978-3-319-79063-3_6 (cited on page 80).

[Ope] OpenBenchmarking.org. https://openbenchmarking.org/processors (cited on page 46).

[PLK+25] S. Park et al. 30.1 A 28gb/mm^{2}4xx-layer 1tb 3b/cell wf-bonding 3d-nand flash with 5.6gb/s/pin ios. In *IEEE International Solid-State Circuits Conference, ISSCC 2025, San Francisco, CA, USA, February 16-20, 2025*, pages 1–3. IEEE, 2025. URL: https://doi.org/10.1109/ISSCC49661.2025.10904543 (cited on page 83).

[Por] T. Pornin. Bearssl's constant time aes implementation. URL: www.bearssl.org/constanttime.html#aes (cited on page 51).

[Riv26] M. Rivain. AES-based grinding for MPC-in-the-head signatures. Cryptology ePrint Archive, Paper 2026/1625, 2026. URL: https://eprint.iacr.org/2026/1625 (cited on pages 14, 73, 96).

[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, Santa Barbara, CA, USA. Springer, Cham, Switzerland, 2022 (cited on pages 9, 56).

[SS16] P. Schwabe and K. Stoffelen. All the AES you need on cortex-m3 and m4. Cryptology ePrint Archive, Paper 2016/714, 2016. URL: https://eprint.iacr.org/2016/714 (cited on page 51).

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Report 2021/582, 2021. URL: https://eprint.iacr.org/2021/582 (cited on pages 14, 95).

[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, Virtual Event. Springer, Cham, Switzerland, 2022 (cited on page 4).

[Tha23] J. Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2023. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf (cited on page 4).

[Wik] Wikipedia. AVX-512. Note: Intel fusing off AVX-512 support on Alder Lake. https://en.wikipedia.org/wiki/AVX-512#endnote_adl-avx512-note (cited on page 47).

[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. URL: https://doi.org/10.1007/978-3-540-27800-9_24 (cited on page 79).

[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, Virtual Event, Republic of Korea. ACM Press, 2021 (cited on page 5).