IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–26. DOI:XXXXXXXX

MQ on my Hardware: Performance Analysis of MQOM on FPGA

Stelios Manasidis[1,2], Quinten Norga[1], Suparna Kundu[1] and Ingrid Verbauwhede[1]

[1] COSIC, KU Leuven, Leuven, Belgium,
{quinten.norga,suparna.kundu,ingrid.verbauwhede}@esat.kuleuven.be
[2] ETRO, Vrije Universiteit Brussel, Brussels, Belgium,
stelios.manasidis@vub.be

Abstract. Recent algorithmic advancements in the Multi-Party Computation-in-the-Head (MPCitH) paradigm have resulted in more efficient post-quantum digital signature schemes. MQOM is an MPCitH-based digital signature scheme and candidate in the ongoing NIST Post-Quantum Cryptography (PQC) standardization effort, offering performance competitive with lattice- and multivariate-based schemes in software. In this work, we develop a dedicated hardware accelerator for MQOM and analyze the impact of recent algorithmic modifications on hardware performance. This is achieved through the careful co-design of high-throughput symmetric primitive engines and highly-optimized polynomial arithmetic cores, minimizing stalling and supporting entirely on-the-fly computations of all polynomial arithmetic. As a result, no intermediate buffers are required and re-computation or sampling is avoided. Secondly, we analyze MQOM's use of correlated GGM trees for generating MPC party shares, which reduce computational cost at the cost of increased signature size. We observe that this choice leads to increased design flexibility and significantly reduces on-chip memory requirements for hardware designs. Furthermore, MQOM proposes several parameter sets per security level. We analyze the impact of different MQOM parameter sets on hardware cost and performance. Our design with NIST L1 parameters only requires 15 812/9 384 LUTs/FFs and 4.5 BRAMs on FPGA, while performing the signature generation in 0.46 ms and signature verification in 0.38 ms. Compared to state-of-the-art hardware implementations of other MPCitH-based DSAs, we improve the area-time-product (ATP) by a factor 3.5× up to 414.8×. Compared to the lattice-based ML-DSA scheme, our MQOM hardware design is only outperformed by a factor 1.5×. Our results show that MQOM and the correlated GGM tree structure are hardware-friendly designs, leading to one of the highest HW-vs-SW speedup ratios among similar PQC DSAs, while also attaining the smallest on-chip memory footprint among high-performance hardware implementations.

Keywords: Post-Quantum Cryptography · Digital Signatures · MPC-in-the-Head · MQOM · RTL · FPGA

1 Introduction

Recent advances in quantum computing pose significant threats to currently deployed public-key cryptosystems [RSA78, Mil85]. In anticipation of the quantum threat, the National Institute of Standards and Technology (NIST) launched their Post-Quantum Cryptography (PQC) standardization effort in 2016 [CJL+16]. This initial process has resulted in the selection (and standardization) of three post-quantum digital signature algorithm (DSA) standards: ML-DSA [Nat24a], FN-DSA [FHK+22], and SLH-DSA [Nat24b]. Since the selected schemes are fundamentally based on lattice or hash-based hard problems, NIST initiated an

Licensed under Creative Commons License CC-BY 4.0.

additional standardization process for digital signatures [Nat23]. In May 2026, NIST selected nine candidates to advance to the third round [ABC+26] of this process: FAEST [BBB+25], HAWK [BBD+25], MAYO [BCC+25], MQOM [BBFR25], QR-UOV [FIH+25], SDitH [MBB+25], SNOVA [WCD+25], SQIsign [AAA+25], and UOV [BCD+25].

The additional signature candidates provide a broader range of security assumptions and performance trade-offs than the first set of PQC standards. Several candidates, including those based on the Multi-Party Computation-in-the-Head (MPCitH) paradigm [IKOS07, FR23, FR25, BBdSG+23], offer attractive bandwidth properties (public key and signature size), at the cost of higher computational complexity. Among the second-round MPCitH candidates, NIST identified MQOM as having the smallest combined public key and signature size at all three targeted security levels among all MPCitH-based candidates, while relying on the relatively stable and conservative hard problem of solving random multivariate quadratic (MQ) systems of equations over finite fields [ABC+26].

The MPCitH framework, originally proposed by Ishai et al. [IKOS07], allows a single prover to perform a Zero-Knowledge Proof (ZKP) of a secret x by locally emulating an N-party MPC protocol "in their head". The verifier then checks the consistency of only a subset of these parties, commonly $N-1$, which yields a non-negligible soundness error. To obtain the target security level, the MPC protocol is repeated τ times in parallel before being compiled into a non-interactive signature scheme using the Fiat-Shamir heuristic.

MPCitH-based schemes rely on Goldreich-Goldwasser-Micali (GGM) trees [GGM86] to generate the MPC shares for each (parallel) repetition, by extending a root seed until its lowest-level nodes (called *leaves* or *leaf nodes*) are obtained, by repeatedly applying a Pseudorandom Generator (PRG) to each node. During the final signature generation step, *all except one* hidden leaf of *each* GGM tree is revealed to the verifier by including its *sibling path* in the signature. The *one-tree* optimization proposed by Baum et al. [BBM+24], used by most MPCitH-based schemes, instead uses a shared larger GGM tree of $\tau \cdot N$ leaves for all parallel MPC protocol repetitions, significantly reducing signature size. MQOM instead relies on the *correlated* GGM (cGGM) tree optimization proposed by Guo et al. [GYW+23], which is less complex and requires expanding a separate GGM tree per protocol repetition.

Related Work A significant amount of research on optimizations of software implementations of MQOM has been performed. First of all, optimized vectorized implementations for x86 processors have been developed, including AVX2-, AVX-512-, and GFNI-based arithmetic [BBFR25]. Secondly, memory-aware implementations for resource-constrained microcontrollers were proposed in [BF26], which include pre-computation techniques to improve the execution-time and memory trade-offs.

In contrast, specialized hardware acceleration of modern MPCitH-based DSAs, including MQOM, remains limited despite their reliance on symmetric primitives and structured finite-field arithmetic. Hardware implementations have been proposed for Picnic [KRR+20] and SDitH [DHSY24, HHH+26], showing that MPCitH-based signatures can benefit from dedicated acceleration, but in some cases also exposing high latency and/or area costs compared to more established PQC DSA hardware designs. Interestingly, modern MPCitH-based schemes, such as Mirath, rely on the *One-GGM-Tree* optimization [BBM+24], causing highly complex control circuitry to bottleneck the overall performance in hardware [MNKV26]. To the best of our knowledge, no prior work on the hardware acceleration of MQOM and its *correlated GGM trees* [GYW+23], which perform well in software, exists.

Contributions In this work, we propose an efficient hardware implementation for MQOM [BBFR25], a promising candidate in the ongoing NIST PQC DSA standardization effort. More specifically:

1. We analyze the performance of correlated GGM trees, as used by MQOM, in hardware. We observe that the separate per-repetition GGM trees enable low-complexity binary

tree expansion, efficient buffer reuse, and reduced control complexity compared with the more tightly coupled *one-tree* approach, used by other MPCitH schemes.

2. We design a unified hardware accelerator for MQOM, supporting the `KeyGen`, `Sign`, and `Verify` routines.[1] Our architecture combines high-throughput symmetric primitive engines with highly-optimized on the fly polynomial evaluations, and memory-efficient GGM-tree buffering. The proposed arithmetic datapath enables entirely on-the-fly computation of the polynomial arithmetic, allowing us to significantly reduce total latency and on-chip memory requirements.

3. We carefully analyze the proposed MQOM parameter sets with respect to hardware performance. Specifically for the base-field $\mathbb{F}_{2^4}$ instances, we introduce an area-efficient fused $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit that allows us to narrow the area-footprint gap between the $\mathbb{F}_{2^4}$ base-field and the more hardware-friendly $\mathbb{F}_2$ instances.

4. We evaluate the performance of our design on FPGA and compare it with optimized software and hardware implementations of PQC DSA schemes. We observe that our MQOM design achieves the lowest ATP among all MPCitH-based designs in the literature, highlighting the suitability of MQOM for hardware. Through optimized scheduling on the proposed architecture, the workloads in the symmetric primitive engines and the arithmetic datapaths are balanced, ensuring that the instantiated logic remains maximally active and utilized.

2 Preliminaries

2.1 Notations

$\lambda \in \{128,192,256\}$ always indicates the security parameter/level of the scheme. Matrices are denoted by uppercase letters (e.g., A), while scalars and vectors are denoted by lowercase letters (e.g., x). Bitwise XOR is denoted by $\oplus$. We use the notation T to denote the transpose of a matrix or vector (i.e. t^{T}). We use bracket notation to denote individual elements of a matrix or vector. For example, $t[j]$ means the j-th element of vector t, while $A[i,j]$ denotes the i-th row, j-th column element of matrix A.

All arithmetic operations are performed over the base field $\mathbb{F}_q$, where $q \in \{2, 16, 256\}$, or the extension field $\mathbb{F}_{q^\mu}$. In the following, we denote the base field by $\mathbb{F} := \mathbb{F}_q$ and its degree-μ extension field by $\mathbb{K} := \mathbb{F}_{q^\mu}$.

MQOM also relies on two symmetric primitives, an extendable-output (& hash) function[2] and block cipher[3]:

$$\text{XOF} : (\texttt{in, len}) \in \{0,1\}^* \times \mathbb{N} \mapsto \texttt{out} \in \{0,1\}^{\texttt{len}}$$

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

2.2 Correlated GGM Trees

MQOM uses *correlated* GGM (cGGM) trees [GYW+23] to derive pseudorandomness used in its polynomial computations, by expanding multiple GGM trees from the parent seeds to the leaf nodes. Compared to the *one GGM tree* structure, where one large, merged GGM tree is expanded, implementation complexity is significantly reduced.

The cGGM tree structure (Figure 1) is based on the *half-tree* structure: each parent node (e.g. node[2]) is used to derive two child nodes. The left child (node[4]) is derived using

[1] The source code of our design is available at https://github.com/KULeuven-COSIC/MQOM-in-HW.
[2] SHAKE128 for NIST I, SHAKE256 for NIST III/V
[3] AES-128 for NIST I, Rijndael-256-256 for NIST III/V

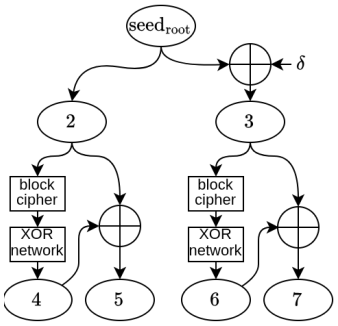

Figure 1: Correlated GGM Tree in MQOM [BBFR25]

a length-preserving PRG primitive (block cipher + XOR network), whereas the second child (node[5]) is derived as the XOR of the left child with the parent node (e.g. node[2] $\oplus$ node[4]). As a result, the computational cost of deriving the leaf nodes in the tree is halved.

In MQOM specifically, an additional *masking* network is applied at the output of the block cipher-based PRG function and the first two child nodes are chosen as the root seed and a corrected root seed [BBFR25], computed as $\text{seed}_{\text{root}} \oplus \delta$. We discuss the implementation of the PRG engine, as well as the XOR network in hardware in Section 4.3.

2.3 MQOM

MQOM (v2.1), referred to as MQOM2 in [BBFR25], is a post-quantum digital signature scheme whose security relies on the hardness of solving random multivariate quadratic systems over finite fields. The most recent specification of MQOM (v2.1) uses the TCitH framework, that is described through the Polynomial Interactive Oracle Proof (PIOP) formalism [BBFR25, Fen24].

A wide variety of MQOM parameters are proposed, including a *short* and *fast* variant, and 3-round (3r) and 5-round (5r) version. The short variant has the smallest communication cost, while the fast variant has lower computational complexity, with $11.3\% - 21.1\%$ increased communication cost. In this work, we target the *fast* variants of NIST security level L1 (see Table 1). This choice does not restrict the applicability of our proposed methodologies and optimizations, which directly generalize to higher security levels and *short* parameter sets (see Section 7). MQOM also proposes a variant over $\mathbb{F}_{16}$, which avoids the larger signatures induced by the larger base field $\mathbb{F}_{256}$, while significantly outperforming the $\mathbb{F}_2$ variant on software platforms [BBFR25].

Table 1: MQOM-fast parameter sets [BBFR25].

MQOM2 Instance	**Security**		**MQ parameters**		**Proof system parameters**					**Comm. cost**	
	NIST	λ	$\lvert\mathbb{F}\rvert$	$m=n$	τ	N	μ	η	w	pk (Bytes)	Sig. (3r/5r) (Bytes)
L1-gf2-fast-3r/5r	Cat. I	128	2	160	17	256	8	20/16	9	52	3 212/3 144
L1-gf16-fast-3r/5r	Cat. I	128	16	56	17	256	2	28/16	9	60	3 484/3 280
L1-gf256-fast-3r/5r	Cat. I	128	256	48	17	256	1	48/16	9	80	4 164/3 620
L3-gf2-fast-3r/5r	Cat. III	192	2	240	27	256	8	30/24	3	78	7 576/7 414
L3-gf16-fast-3r/5r	Cat. III	192	16	84	27	256	2	42/24	3	90	8 224/7 738
L3-gf256-fast-3r/5r	Cat. III	192	256	72	27	256	1	72/24	3	120	9 844/8 548
L5-gf2-fast-3r/5r	Cat. V	256	2	320	36	256	8	40/32	4	104	13 412/13 124
L5-gf16-fast-3r/5r	Cat. V	256	16	116	36	256	2	58/32	4	122	14 708/13 772
L5-gf256-fast-3r/5r	Cat. V	256	256	96	36	256	1	96/32	4	160	17 444/15 140

3 Computational Analysis of MQOM

We present the top-level key generation, signing, and verification routines below (Alg. 1, 3, 6), analyze them and identify their main computational routines.

3.1 Key Generation

The secret key is derived from a uniformly sampled seed, from which the secret MQ solution x and the master equation seed `mseed_eq` are expanded. The public MQ instance is then generated deterministically from `mseed_eq` using `ExpandEquations`, shown in Algorithm 2. This routine expands `mseed_eq` into the packed MQ equations $(\{\hat{A}_i\},\{\hat{b}_i\})$, after which the public vector $\hat{y}$ is computed as $\hat{y}_i = x^\top \hat{A}_i x + \hat{b}_i^\top x$ for all $i \in \{1,2,...,\hat{m}\}$, $\hat{m} = \frac{m}{\mu}$. The arithmetic required to compute the public vector $\hat{y}$ is inexpensive compared with the `Enc` PRG calls within `ExpandEquations`, whose computational cost dominates key generation.

Algorithm 1 `KeyGen()`

Output: a secret key `sk`, a public key `pk`
1: `seed_key` $\leftarrow \{0,1\}^{2\lambda}$
2: $(x$,`mseed_eq`$) \leftarrow$ `Parse`(`XOF`$_0$(`seed_key`,`len`$:=|x|_2+2\lambda$)) ▷ $x \in \mathbb{F}^n$, `mseed_eq` $\in \{0,1\}^{2\lambda}$
3: $(\{\hat{A}_i\},\{\hat{b}_i\}) \leftarrow$ `ExpandEquations`(`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: $\quad \hat{y}_i \leftarrow x^\top \hat{A}_i x + \hat{b}_i^\top x$ ▷ $\hat{y}_i \in \mathbb{K}$
6: $\hat{y} \leftarrow (\hat{y}_1,...,\hat{y}_{\hat{m}})$ ▷ $\hat{y} \in \mathbb{K}^{\hat{m}}$
7: `pk` $\leftarrow$ `Serialize`(`mseed_eq`,$\hat{y}$)
8: `sk` $\leftarrow$ `Serialize`(`pk`,x)
9: **return** (`pk`,`sk`)

The `ExpandEquations` routine is used not only during key generation, but also during signing and verification, where the same packed MQ equations must be regenerated from the public seed. Consequently, this routine is one of the recurring computational components of MQOM and one of the main optimization targets of this work (see Section 4.1).

Algorithm 2 `ExpandEquations`(`mseed_eq`)

Input: a seed key `mseed_eq` $\in \{0,1\}^{2\lambda}$
Output: packed MQ equations $(\{\hat{A}_i\},\{\hat{b}_i\})$
1: Let $n_{\text{feq}} = n + \sum_{j=1}^{n} j$ ▷ Number of field elements for $(\hat{A}_i,\hat{b}_i)$
2: Let $n_{\text{beq}} = n_{\text{feq}} \cdot \frac{\log_2|\mathbb{K}|}{8}$ ▷ Number of PRG bytes for $(\hat{A}_i,\hat{b}_i)$
3: **for** $i=1$ to $\hat{m}$ **do**
4: $\quad$ `seed_eq`$[i-1] \leftarrow$ `XOF`$_1$((`mseed_eq`,`Bits`$_{16}(i-1)$),`len`$:=\lambda$)
5: $\quad$ (`row`$_{i,1}$,...,`row`$_{i,n}$,$\hat{b}_i$) $\leftarrow$ `Parse`(`PRG`$(0^\lambda$,0,`seed_eq`$[i-1]$,n_{beq})) ▷ `row`$_{i,j} \in \mathbb{K}^j$, $\hat{b}_i \in \mathbb{K}^n$
6: $\quad$ **for** $j=1$ to n **do**
7: $\quad\quad \hat{A}_{i,j} \leftarrow$ (`row`$_{i,j} \,\|\, 0_{\mathbb{K}^{n-j}}$) ▷ $\hat{A}_{i,j} \in \mathbb{K}^n$, jth row of $\hat{A}_i$
8: $\quad \hat{A}_i \leftarrow (\hat{A}_{i,1},...,\hat{A}_{i,n})$ ▷ $\hat{A}_i \in \mathbb{K}^{n\times n}$
9: **return** $(\{\hat{A}_i\},\{\hat{b}_i\})$

3.2 Signature Generation

The generated signature proves knowledge of a secret MQ solution $x \in \mathbb{F}^n$ such that, for the public packed MQ instance $(\{\hat{A}_i\},\{\hat{b}_i\},\hat{y})$, it holds that $\hat{y}_i = x^\top \hat{A}_i x + \hat{b}_i^\top x$ for all $i \in \{1,2,...,\hat{m}\}$.

The two main computational components of signing are the batch line commitment (`BLC.Commit`) and the evaluation of the MQ relation over the committed polynomials (`ComputePAlpha`). First, `BLC.Commit` (Alg. 4) expands the master seed `mseed` into τ root

Algorithm 3 `Sign(sk, msg)`

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

```
(pk,x) = Parse(sk)
(mseed_eq,ŷ) = Parse(pk)
mseed ← {0,1}^λ
salt ← {0,1}^λ
msg_hash ← Hash_2(msg)
(com_1,key,x_0,u_0,u_1) ← BLC.Commit(mseed,salt,x)
(α_0,α_1) ← ComputePAlpha(com_1,x_0,u_0,u_1,x,mseed_eq)
com_2 ← Hash_3(α_0,α_1)
hash ← Hash_4(pk,com_1,com_2,msg_hash)
(i*,nonce) ← SampleChallenge(hash)
opening ← BLC.Open(key,i*)
return sig := Serialize(salt,com_1,com_2,α_1,opening,nonce)
```

seeds and, for each repetition $e \in [0,\tau-1]$, derives an N-leaf GGM tree, where the leaves correspond to the N evaluation points $\Omega = \{\omega_0,\ldots,\omega_{N-1}\}$. Its computational cost is therefore dominated by the large number of `Enc` PRG calls required for GGM-tree expansion, `SeedCommit`, and leaf-seed expansion. Here, `GGMTree.Expand` returns the pair (`node`$[e]$,`lseed`$[e]$). The array `node`$[e]$ contains $2N-2$ node seeds, indexed from 2 to $2N-1$, each of length λ bits. The array `lseed`$[e]$ contains the N leaf seeds, with `lseed`$[e][i]$ = `node`$[e][N+i]$ for $i \in [0,N-1]$. `TweakSalt` modifies the public signature salt so that different operations and repetitions use different block-cipher keys. For example, `TweakSalt(salt,0,e,0)` gives the key used for leaf-seed commitments in repetition e. `SeedCommit` uses `Enc` to produce a 2λ-bit commitment to a leaf seed. The array `ls_com`$[e]$ contains these N commitments, where `ls_com`$[e][i]$ commits to `lseed`$[e][i]$. This array is hashed to obtain `hash_ls_com`$[e]$. The precise definitions of `GGMTree.Expand`, `TweakSalt`, and `SeedCommit` are given in Algorithms 14, 17, and 19 of the MQOM v2.1 specification [BBFR25], respectively.

Second, `ComputePAlpha` regenerates the packed MQ equations $(\{\hat{A}_i\}, \{\hat{b}_i\})$ from `mseed_eq` using `ExpandEquations`, and combines them with the committed polynomial coefficients to compute (α_0,α_1). `ComputePAlpha` is computationally intensive for two reasons: `ExpandEquations` requires multiple `Enc` PRG calls, while the subsequent polynomial evaluation requires multiple finite-field operations. Unlike the computation of $\hat{y}$ during key generation, this arithmetic is performed over all τ protocol repetitions. The relative cost of `BLC.Commit` and `ComputePAlpha` varies substantially across the different parameter sets of MQOM (see Section 7.1).

Finally, MQOM applies grinding when sampling the challenge indices i^* (Alg. 5). Grinding reduces the required number of parallel repetitions to $(2/N)^\tau \leq \frac{1}{2^{(\lambda-w)}}$ by searching for a challenge whose additional w-bit value is zero, at the cost of 2^w challenge-sampling trials on average. As a stochastic procedure, grinding introduces runtime variability into the signing routine. However, as grinding depends only on public data, this timing variability does not leak any secret information.

For each repetition, the leaf indexed by i^* remains hidden, while the signature contains the sibling path needed to reconstruct all other leaf seeds, together with the commitment to the hidden leaf seed.

3.3 Signature Verification

The verifier first recomputes the message hash and transcript hash, derives the challenge indexes i^* using the provided `nonce`, and checks the grinding condition by verifying that the sampled w-bit value is zero.

In `BLC.Eval`, which is structurally similar to `BLC.Commit`, the sibling paths contained in the signature are used to reconstruct all opened GGM-tree leaves, and their commitments

Algorithm 4 `BLC.Commit(mseed,salt,`x`)`

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

1: $(\texttt{rseed}[0],\ldots,\texttt{rseed}[\tau-1]) \leftarrow \texttt{Parse}(\texttt{PRG}(0^\lambda,0,\texttt{mseed},\tau\cdot\lambda))$
2: $\delta \leftarrow \texttt{FirstBits}_\lambda(x)$ ▷ $\delta\in\{0,1\}^\lambda$
3: **for** $e=0$ to $\tau-1$ **do**
4: $\quad(\texttt{node}[e],\texttt{lseed}[e]) \leftarrow \texttt{GGMTree.Expand}(\texttt{salt},\texttt{rseed}[e],e,\delta)$
5: $\quad\texttt{tweaked_salt} \leftarrow \texttt{TweakSalt}(\texttt{salt},0,e,0)$
6: $\quad$**for** $i=0$ to $N-1$ **do**
7: $\quad\quad\texttt{ls_com}[e][i] \leftarrow \texttt{SeedCommit}(\texttt{tweaked_salt},\texttt{lseed}[e][i])$
8: $\quad\quad\texttt{exp}[e][i] \leftarrow \texttt{PRG}(\texttt{salt},e,\texttt{lseed}[e][i],|x|_8+|u|_8-\lambda/8)$
9: $\quad\quad(\bar{x}_i,\bar{u}_i) \leftarrow \texttt{Parse}(\texttt{lseed}[e][i]\,\|\,\texttt{exp}[e][i])$ ▷ $\bar{x}_i\in\mathbb{F}^n$, $\bar{u}_i\in\mathbb{K}^\eta$
10: $\quad\texttt{hash_ls_com}[e] \leftarrow \texttt{Hash}_6(\texttt{ls_com}[e])$
$\quad$▷ Compute $P_u = u_0[e]+u_1[e]\cdot X = \sum_{i=0}^{N-1}\bar{u}_i\cdot(X-\omega_i)$
11: $\quad u_0[e] \leftarrow -\sum_{i=0}^{N-1}\omega_i\cdot\bar{u}_i$ ▷ $u_0[e]\in\mathbb{K}^\eta$
12: $\quad u_1[e] \leftarrow \sum_{i=0}^{N-1}\bar{u}_i$ ▷ $u_1[e]\in\mathbb{K}^\eta$
$\quad$▷ Compute $P_x = x_0[e]+x\cdot X = \Delta_x[e]\cdot X+\sum_{i=0}^{N-1}\bar{x}_i\cdot(X-\omega_i)$
13: $\quad x_0[e] \leftarrow -\sum_{i=0}^{N-1}\omega_i\cdot\bar{x}_i$ ▷ $x_0[e]\in\mathbb{K}^n$
14: $\quad\Delta_x[e] \leftarrow x-\sum_{i=0}^{N-1}\bar{x}_i$ ▷ $\Delta_x[e]\in\mathbb{F}^n$
15: $\quad\Delta_x^{(1)}[e] \leftarrow \texttt{NextBits}_\lambda(\Delta_x[e])$ ▷ $\Delta_x^{(1)}[e]\in\{0,1\}^{|x|_2-\lambda}$
16: $\texttt{com}_1 \leftarrow \texttt{Hash}_7(\texttt{hash_ls_com},\Delta_x^{(1)})$
17: $\texttt{key} \leftarrow \texttt{Serialize}(\texttt{node},\texttt{ls_com},\Delta_x^{(1)})$
18: **return** $(\texttt{com}_1,\texttt{key},x_0,u_0,u_1)$

Algorithm 5 `SampleChallenge(hash)`

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

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

are checked against $\texttt{com}_1$. It produces the evaluations $v_x = P_x(r)$ and $v_u = P_u(r)$ at the challenged point $r = \omega_{i^*[e]}$ for each repetition. Finally, the verifier recomputes the coefficient array α_0 using `RecomputePAlpha`, which is very similar to `ComputePAlpha`. Thus, verification exhibits the same computational balance as signing, between BLC processing and packed-equation evaluation (including cGGM expansion). The hardware implications of this balance across the different MQOM parameter sets are discussed in Section 7.1.

4 Proposed Optimizations

Through careful analysis of MQOM, we obtain that its performance is mainly dominated by repeated `Enc` PRG calls during both `ExpandEquations` and **cGGM seed expansion**, and **polynomial evaluations/arithmetic** in `BLC.Commit`, `BLC.Eval` and `ComputePAlpha`. This section presents the main algorithmic optimizations that we employ in our MQOM hardware accelerator, including on-the-fly polynomial arithmetic computations, and memory-efficient GGM tree buffering.

Algorithm 6 `Verify(pk,msg,sig)`

Input: a public key `pk`, a message `msg`, a signature `sig`
Output: `True` or `False`

1: $(\texttt{mseed_eq},\hat{y}) = \texttt{Parse}(\texttt{pk})$
2: $(\texttt{salt},\texttt{com}_1,\texttt{com}_2,\alpha_1,\texttt{opening},\texttt{nonce}) = \texttt{Parse}(\texttt{sig})$
3: $\texttt{msg_hash} \leftarrow \texttt{Hash}_2(\texttt{msg})$
4: $\texttt{hash} \leftarrow \texttt{Hash}_4(\texttt{pk},\texttt{com}_1,\texttt{com}_2,\texttt{msg_hash})$
5: $(i^*,\texttt{val}) \leftarrow \texttt{Parse}(\texttt{XOF}_5((\texttt{hash},\texttt{nonce}),\texttt{len}:=\tau\cdot\log_2(N)+w))$
6: **if** $\texttt{val} \neq 0$ **then**
7: **return** `False`
8: $(\texttt{ret},\texttt{x_eval},\texttt{u_eval}) \leftarrow \texttt{BLC.Eval}(\texttt{salt},\texttt{com}_1,\texttt{opening},i^*)$
9: **if** $\texttt{ret} \neq \texttt{True}$ **then**
10: **return** `False`
11: $\alpha_0 \leftarrow \texttt{RecomputePAlpha}(\texttt{com}_1,\alpha_1,i^*,\texttt{x_eval},\texttt{u_eval},\texttt{mseed_eq},\hat{y})$
12: $\texttt{com}_2' \leftarrow \texttt{Hash}_3(\alpha_0,\alpha_1)$
13: **if** $\texttt{com}_2' \neq \texttt{com}_2$ **then**
14: **return** `False`
15: **return** `True`

4.1 Streaming the MQ Coefficients

The `ExpandEquations` routine (Alg. 2) expands the public seed `mseed_eq` into the packed MQ equations $(\{\hat{A}_i\},\{\hat{b}_i\})$. For each packed equation, the PRG output contains the nonzero entries of the triangular matrix $\hat{A}_i$, followed by the entries of the vector $\hat{b}_i$:

$$\hat{A}_i[1,1],\ \hat{A}_i[2,1],\ \hat{A}_i[2,2],\ldots,\hat{A}_i[n,1],\ldots,\hat{A}_i[n,n],\ \hat{b}_i[1],\ldots,\hat{b}_i[n].$$

where the nonzero entries of $\hat{A}_i$ are denoted by $\hat{A}_i[j,k]$ for $1 \leq k \leq j \leq n$. In the provided software implementations, these sampled elements are inserted into an $n \times n$ matrix and the remaining entries are filled with zeros:

$$\hat{A}_i = \begin{bmatrix} \hat{A}_i[1,1] & 0 & \cdots & 0 \\ \hat{A}_i[2,1] & \hat{A}_i[2,2] & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \hat{A}_i[n,1] & \hat{A}_i[n,2] & \cdots & \hat{A}_i[n,n] \end{bmatrix}.$$

This representation is convenient for expressing the MQ relation as a standard matrix computation, but it is inefficient for hardware. Any dedicated hardware datapath that fully materializes $\hat{A}_i$ would spend the majority of the cycles it spends on the first rows on padding zeros. This would inevitably lead to a significant stalling of the `Enc` PRG primitive and introduce significant latency overhead. We therefore choose to directly work with the triangular coefficient stream of $\hat{A}_i$ throughout the polynomial arithmetic, in key generation, signature generation and signature verification. In signing and verification, the same coefficient stream is consumed by all τ repetitions in parallel. In the following subsection we describe how the sequential PRG output is directly combined with other terms/vectors, avoiding repeated expansion of the same $(\hat{A}_i,\hat{b}_i)$ data.

4.2 On-the-fly Polynomial Computation

The `ComputePAlpha` routine computes the polynomial P_z through the intermediate polynomial $P_t = t_0 + t_1 \cdot X = \hat{A}_i \cdot P_x + \hat{b}_i \cdot X$. For each protocol repetition $e \in [0,\tau-1]$ and packed equation index $i \in [1,\hat{m}]$, the coefficient vectors t_0 and t_1 are computed as

$$t_0 = \hat{A}_i \cdot x_0 \quad \text{and} \quad t_1 = \hat{A}_i \cdot x + \hat{b}_i$$

respectively. The coefficients of P_z are then computed as

$$z_{0,i} = t_0^{\mathsf{T}} \cdot x_0 \quad \text{and} \quad z_{1,i} = t_0^{\mathsf{T}} \cdot x + t_1^{\mathsf{T}} \cdot x_0.$$

A naïve implementation would require storing the intermediate vectors t_0 and t_1 and reading them back to compute $z_{0,i}$ and $z_{1,i}$. Instead, we propose a hardware-friendly version of `ComputePAlpha` (Alg. 7), in which the above expressions are expanded such that the coefficients of P_z are directly accumulated from the packed equation stream (Section 4.1). Here, the loop order is adapted such that each packed equation $(\hat{A}_i,\hat{b}_i)$ is expanded once and used for all τ protocol repetitions in parallel. More specifically, for each packed equation i and repetition e, we directly compute:

$$z_{0,i} = x_0^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0 = \sum_{j=1}^{n} \sum_{k=1}^{j} \hat{A}_i[j,k] x_0[j] x_0[k],$$

$$z_{1,i} = x^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0 + \hat{b}_i^{\mathsf{T}} \cdot x_0 + x_0^{\mathsf{T}} \cdot \hat{A}_i \cdot x = \sum_{j=1}^{n} \sum_{k=1}^{j} \hat{A}_i[j,k](x[j]x_0[k] + x_0[j]x[k]) + \sum_{j=1}^{n} \hat{b}_i[j] x_0[j].$$

Every coefficient of $\hat{A}_i$ or $\hat{b}_i$ contributes to $z_{0,i}$ and $z_{1,i}$ as soon as it becomes available and is discarded immediately after, while sub-terms that are independent of the parallel repetition index e are shared across all parallel computations.

Algorithm 7 HW-optimized `ComputePAlpha`(`com`,x_0,u_0,u_1,x,`mseed_eq`)

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

1: **if** `batching` **then** ▷ `batching` = `True` ⇒ 5-round variant
2: $\quad\Gamma \in \mathbb{K}^{\eta \times \hat{m}} \leftarrow$ `Parse`(`XOF`$_8$(`com`,`len`$:=\eta \cdot \hat{m} \cdot \log_2(|\mathbb{K}|)$))
3: **else** ▷ `batching` = `False` ⇒ 3-round variant
4: $\quad\Gamma \leftarrow I_\eta$ ▷ Identity matrix $I_\eta \in \mathbb{K}^{\eta \times \eta}$ with $\eta = \hat{m}$
5: **for** $e = 0$ to $\tau - 1$ **do**
6: $\quad\alpha_0[e] \leftarrow u_0[e]$ ▷ $\alpha_0[e] \in \mathbb{K}^\eta$
7: $\quad\alpha_1[e] \leftarrow u_1[e]$ ▷ $\alpha_1[e] \in \mathbb{K}^\eta$
8: $(\{\hat{A}_i\},\{\hat{b}_i\}) \leftarrow$ `ExpandEquations`(`mseed_eq`) ▷ Expanded coefficients streamed on the fly
9: **for** $i = 1$ to $\hat{m}$ **do**
10: $\quad$**for** $e = 0$ to $\tau - 1$ **(in parallel) do**
11: $\quad\quad z_{0,i} \leftarrow x_0[e]^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0[e]$ ▷ $z_{0,i} \in \mathbb{K}$
12: $\quad\quad z_{1,i} \leftarrow x^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0[e] + \hat{b}_i^{\mathsf{T}} \cdot x_0[e] + x_0[e]^{\mathsf{T}} \cdot \hat{A}_i \cdot x$ ▷ $z_{1,i} \in \mathbb{K}$
13: $\quad\quad \alpha_0[e] \leftarrow \alpha_0[e] + \Gamma_{:,i} \cdot z_{0,i}$ ▷ Scale the i^{th} column of Γ by $z_{0,i}$ & update $\alpha_0[e]$
14: $\quad\quad \alpha_1[e] \leftarrow \alpha_1[e] + \Gamma_{:,i} \cdot z_{1,i}$
15: **return** (α_0,α_1)

The same ideas are applied to key generation and signature verification, for which a similar computational structure is required. During key generation, we compute the public vector $\hat{y}$ as

$$\hat{y}_i = x^{\mathsf{T}} \cdot \hat{A}_i \cdot x + \hat{b}_i^{\mathsf{T}} \cdot x = \sum_{j=1}^{n} \sum_{k=1}^{j} \hat{A}_i[j,k] x[j] x[k] + \sum_{j=1}^{n} \hat{b}_i[j] x[j].$$

During verification, we compute

$$v_{z,i} = v_x^{\mathsf{T}} \cdot \hat{A}_i \cdot v_x + r_e \cdot \hat{b}_i^{\mathsf{T}} \cdot v_x - r_e^2 \cdot \hat{y}_i = \sum_{j=1}^{n} \sum_{k=1}^{j} \hat{A}_i[j,k] v_x[j] v_x[k] + r_e \sum_{j=1}^{n} \hat{b}_i[j] v_x[j] - r_e^2 \hat{y}_i.$$

where $r_e = \omega_{i^*[e]}$ and $v_x =$ `x_eval`$[e]$. We propose a single, unified polynomial computation core which can be used during all MQOM routines (see Section 5.3), instead of instantiating three separate datapaths.

The choice between 3-round (3r) and 5-round (5r) variants has an impact on `ComputePAlpha`. In the 3r variant, batching is disabled and Γ is set to the identity matrix, so

the prover sends the full polynomial P_z, masked by P_u. In the 5r variant, an additional prover-verifier interaction round is used to derive a random matrix $\Gamma \in \mathbb{K}^{\eta \times \hat{m}}$ with $\eta < \hat{m}$. The prover then sends only the η random linear combinations $\Gamma \cdot P_z$, masked by P_u, instead of the full $\hat{m}$-element polynomial. This reduces the signature size, at the cost of a more complex proof structure and the additional arithmetic required to compute the products $\Gamma \cdot z_0$ and $\Gamma \cdot z_1$.

4.3 Memory-Efficient Correlated GGM Tree Expansion

Previous work on hardware acceleration of MPCitH-based schemes, which rely on the one-tree approach, has shown that memory-efficient GGM tree expansion requires balancing significant recomputation and/or complicated scheduling approaches [MNKV26, HHH+26].

The (correlated) GGM tree expansion in MQOM, which uses a separate GGM tree for each MPC repetition, is described in Algorithm 4 (`BLC.Commit`). A naïve implementation would store the full binary GGM tree for every repetition (τ in total), requiring storage for $2(N-1)\tau$ nodes. However, we observe that the approach taken by MQOM offers more implementation flexibility and reduces the required on-chip memory size. In this work, we use a single tree buffer capable of storing up to N (leaf) nodes, and is reused across all τ repetitions. This is sufficient as obtaining each level in the tree only requires the nodes of the previous level. Using this N-node runtime buffer allows us to use the same low-complexity binary GGM-tree expansion used in the reference software implementation, while reducing the on-chip memory required for GGM-tree expansion by a factor of $\sim 2\tau$. This is especially useful for the *fast* parameter sets, for which $N = 256$ and $\tau \in \{17,27,36\}$.

For the *short* variants of MQOM, where $N = 2048$ and $\tau \in \{12,18,25\}$, the size of each GGM tree is increased so that fewer protocol repetitions can be performed [BBFR25]. In that case, further optimizations can be used to keep the on-chip memory footprint low. In particular, partitioning each repetition's GGM tree into $N_{\text{short}}/N_{\text{fast}} = 8$ smaller sub-trees would allow keeping the same memory footprint for a small overhead in implementation complexity.

The disadvantage of not permanently storing all τ GGM trees is a small recomputation during the opening phase. After the challenge indexes i^* are known, the signer must recover the sibling path of the hidden leaf in each repetition. For a GGM tree of N leaves, this requires $\log_2 N - 1$ additional calls to the `Enc` PRG primitive to regenerate the sibling path, plus two `Enc` PRG calls to recompute the commitment of the hidden leaf. Thus, the total recomputation overhead is $(\log_2 N + 1) \cdot \tau$ additional `Enc` calls, which is negligible compared with the total number of `Enc` calls (on the order of $N \cdot \tau$).

5 Hardware Design

In this section, we provide a detailed description of our unified architecture and each sub-module of our unified MQOM hardware implementation, which supports all MQOM routines.

5.1 High-Level Architecture

Fig. 2 shows the top-level architecture of the proposed MQOM accelerator. The design is centered around the two symmetric primitive engines (SHAKE & Enc PRG) and the polynomial arithmetic datapath. The `Enc` PRG engine generates GGM-tree nodes, leaf commitments, and packed-equation coefficients, while the SHAKE engine is used for hashing, challenge generation, and grinding. The arithmetic datapath consists of the `BLC.FoldingComp`, `MatSeq2Triang`, and `ComputePoly` modules.

The `GetSiblingPath` module is used during `BLC.Open` in signing and during verification. It stores the challenge indexes i^* for all τ protocol repetitions and provides them to the other modules at runtime. It also uses these indices to compute the sibling path of the hidden leaf for each repetition, which is used by the `Enc` PRG to build the signature during `BLC.Open` and

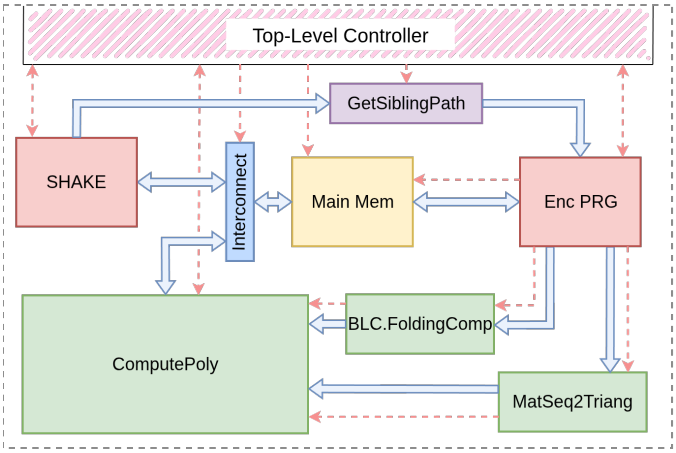

Figure 2: Top-level architecture of our MQOM hardware accelerator. Solid blue arrows represent data streams, while dashed red arrows represent control, status, and metadata signals.

for the partial GGM tree re-expansion during `BLC.Eval`. Due to the choice of *correlated GGM trees* in MQOM, computing the sibling path of each separate GGM tree only takes 10-12 clock cycles. By detaching this functionality from the main FSM, overall complexity is reduced.

5.1.1 Memory Architecture

The accelerator uses a single true dual-port (TDP) memory instance as its main storage. This memory is used across all routines and stores both the secret and public keys, the signature, and intermediate values such as GGM-tree nodes and commitments. We implement it as an asymmetric TDP memory: the first (right) port (connected to the `Enc` PRG) is λ bits wide, which allows us to fix the `Enc` PRG's internal control logic for all security levels by making memory access independent of the security parameter λ. We fix the width of the second (left) port (shared by the SHAKE engine and the polynomial datapath) to 64 bits for all instances. The top-level controller arbitrates access to this port according to the active routine and operation schedule.

5.2 High-Throughput Symmetric Primitive Engines

We now describe the SHAKE and Enc PRG cores, which implement the XOF and Enc symmetric primitives in MQOM (see Section 3).

5.2.1 Grinding SHAKE Engine

Overall, the SHAKE primitive is used sparingly in MQOM and most of its operational cycles can be masked by the operation of the `Enc` PRG, as will be shown in Section 6. Hence, we instantiate a single high-throughput SHAKE engine and reuse it across all routines and subroutines of MQOM. At synthesis time, the SHAKE engine is configured to support one functionality (e.g. SHAKE128 for NIST L1), significantly decreasing both the area footprint and the complexity of our SHAKE engine.

The SHAKE engine is also used for *grinding* during the signing procedure (Alg. 3), in `SampleChallenge`. The grinding operation is equivalent to performing a zero-check on part of the SHAKE output. Due to its highly sequential nature, the `SampleChallenge` subroutine introduces a significant latency overhead (about $20\% - 35\%$) for parameter sets where the grinding parameter w is set to a relatively large value. We observe that in a naive implementation of `SampleChallenge`, using a generic SHAKE engine, data and (control) commands

are repeatedly being sent between the `XOF` output and top-level FSM, where the check is performed, causing up to 40% of the total cycles being contributed by communication overhead.

Instead, we adapt our SHAKE engine so that grinding is fully implemented *internally* in the SHAKE engine. A simple comparator is added to the SHAKE engine, and closely integrated with the SHAKE (Keccak) state. Once the permutation is completed, the comparator checks if the grinding vector (located at a fixed position in the SHAKE state) is zero, before indicating to the top-level controller the computation is finished, alongside sending back the valid challenge. If the grinding vector is not zero, the nonce is increased internally in the SHAKE engine and the `SampleChallenge` is repeated, until a zero vector is obtained, without any involvement from the top-level FSM. This allows us to reduce the latency overhead of `SampleChallenge` by 30-40%, without increasing the area footprint, since the optimization only affects control logic.

5.2.2 Encryption PRG Engine

As discussed in Sections 2 and 4, a significant part of MQOM's computational cost comes from multiple repeated calls to the `Enc` PRG primitive. As a result, our goal is to design an `Enc` PRG engine that has the highest throughput that our architecture (arithmetic datapath) can support, without stalling. This is achieved through an AES/Rijndael encryption core that performs one AES/Rijndael round per clock cycle, paired with an on-the-fly key scheduler that provides a new round key at every cycle.

MQOM avoids the overhead of repeated key expansion by using the same block-cipher `key` (derived from a public salt) for multiple `Enc` PRG calls, where the plaintext is the parent `seed`. This allows us to significantly increase the throughput of our `Enc` PRG engine by using multiple encryption *cores*, but keeping only one shared key scheduler to keep the area of our `Enc` PRG engine minimal (see Figure 3). To avoid trivial decryption-based attacks, MQOM also uses a small feed-forward XOR network at the output of the encryption primitive to derive both child nodes, which is efficient to implement in hardware.

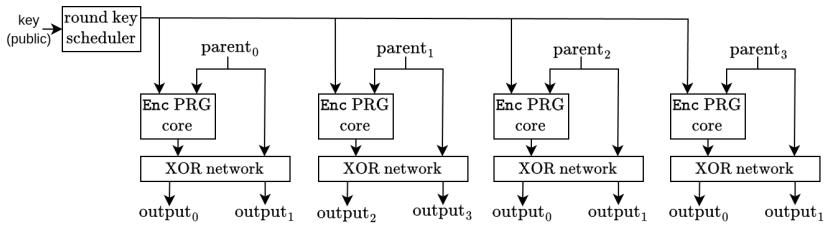

Figure 3: Proposed `Enc` PRG Engine.

Optimal Core Configuration The SHAKE engine limits the number of `Enc` cores that can be used effectively during `BLC.Commit`, since it must absorb the generated leaf commitments without stalling the `Enc` PRG engine. To keep the implementation complexity manageable, we restrict the design space to power-of-two `Enc` core options. Using only one or two parallel `Enc` cores slightly reduces the overall area size, while substantially increasing latency and underutilizing the remaining datapath as the PRG engine causes stalls in other modules. For all considered parameter sets except MQOM-L1-gf16-fast-r3, the commitment generation can be interleaved with the cGGM tree expansion and leaf generation, by instantiating four `Enc` cores in our PRG engine. Although MQOM-L1-gf16-fast-r3 enables instantiating eight cores without being stalled by the SHAKE engine, we retain a fixed architecture with four parallel `Enc` cores across all implemented MQOM instances.

5.3 Optimized Polynomial Arithmetic Datapath

We now describe the modules in the arithmetic datapath that perform the polynomial evaluations in MQOM.

5.3.1 BLC.FoldingComp

Our `BLC.FoldingComp` module implements the arithmetic of `BLC.Commit` (Alg. 4) and `BLC.Eval` [BBFR25, Alg. 12]. For each protocol repetition e, the module consumes the pseudorandomness generated by the `Enc` PRG and computes the values $x_0[e]$, $u_0[e]$, $u_1[e]$, and $\Delta_x[e]$ (during signing) or `x_eval`$[e]$ and `u_eval`$[e]$ (verification). The $x_0[e]$, $u_0[e]$, $u_1[e]$ vectors (or `x_eval`$[e]$ and `u_eval` during verification) are forwarded to the `ComputePoly` module to be used later during the `ComputePAlpha` (or `RecomputePAlpha`) routines.

5.3.2 MatSeq2Triang

The `MatSeq2Triang` module converts the sequential PRG output of `ExpandEquations` into the coefficient stream consumed by the `ComputePoly` module. The packed equations are generated as a sequence of coefficients for the lower-triangular part of each matrix $\hat{A}_i$, followed by the coefficients of the corresponding vector $\hat{b}_i$. `MatSeq2Triang` uses a small set of counters to track the current row and column index. For every valid coefficient, it emits the coefficient together with the metadata needed by the arithmetic datapath, such as the *end-of-row* flag. This module therefore decouples the `Enc` PRG stream from the subsequent polynomial arithmetic. The resulting stream is consumed immediately by `ComputePoly`, enabling the packed equations to be processed on the fly without storing the expanded matrices or vectors in memory.

5.3.3 ComputePoly

Fig. 4 shows the unified `ComputePoly` datapath for the 3-round variants, implementing the rearranged MQ equation formulations described in Section 4. Instead of instantiating separate datapaths for `KeyGen`, `Sign`, and `Verify`, which would significantly increase the number of expensive multiply-accumulate elements, we use a small number of multiplexers (MUXes) that allow us to reuse the same datapath across all three routines. Highly utilized arithmetic units operating on values that differ per protocol repetition are instantiated τ times in parallel (drawn as stacked elements in Fig. 4). This allows us to process all τ protocol repetitions in parallel while replicating only a part of our arithmetic units (Fig. 4).

The module consumes the packed equation coefficients $(\hat{A}_i, \hat{b}_i)$ on the fly, streamed by the `MatSeq2Triang` unit. The following operations are computed using the unified `ComputePoly` datapath:

- `KeyGen`: the public vector $\hat{y}$ is computed as

$$\hat{y}_i = x^{\mathsf{T}} \cdot \hat{A}_i \cdot x + \hat{b}_i^{\mathsf{T}} \cdot x$$

- `Sign`: we compute the polynomial coefficient vectors z_0 and z_1 for each protocol repetition $e \in [0, \tau-1]$ as

$$z_{0,i} = x_0^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0, \qquad z_{1,i} = x^{\mathsf{T}} \cdot \hat{A}_i \cdot x_0 + \hat{b}_i^{\mathsf{T}} \cdot x_0 + x_0^{\mathsf{T}} \cdot \hat{A}_i \cdot x$$

  and then add them to the masking vectors u_0 and u_1 to obtain α_0 and α_1, respectively. We avoid instantiating very wide adders by adding the u_0 and u_1 to z_0 and z_1 a few elements at a time, as the results are written to the main memory.

- `Verify`: the polynomial coefficient vector α_0 is recomputed by reusing the same datapath with the minor addition of the correction terms $-\hat{y} \cdot r^2$ (to compute v_z)

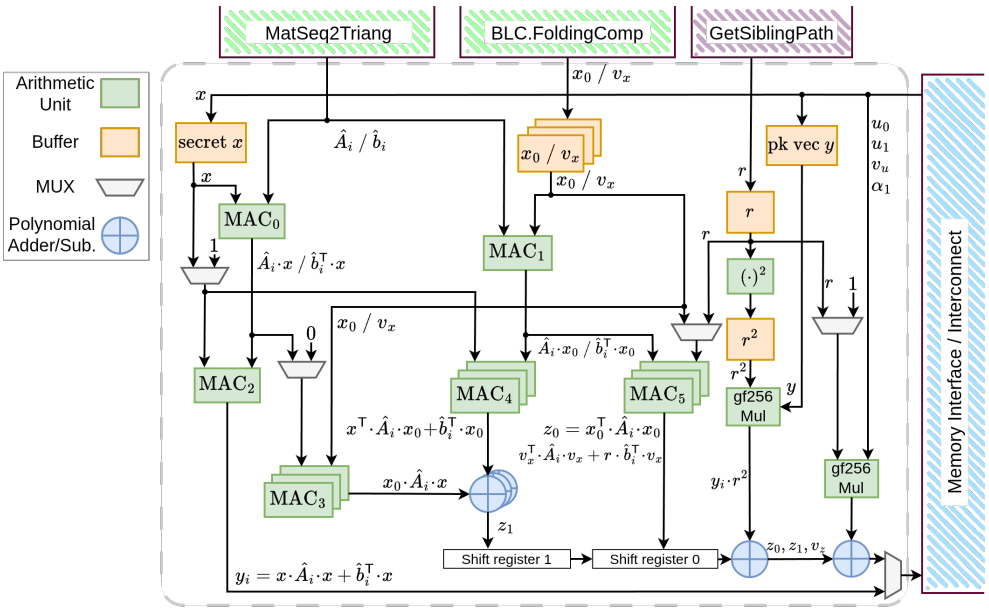

Figure 4: ComputePoly module (3-round version)

and $-\alpha_1 \cdot r$ (to recompute α_0). We note that since all computations are performed over fields of characteristic two, addition and subtraction reduce to the same XOR operation. During signing, the verification-only input r is set to zero, allowing the same datapath to be reused without additional multiplexers.

We note that the 5-round version differs only in the additional batching step, storing the XOF-derived matrix Γ and multiplying the computed vectors z_0, z_1, and v_z by Γ before accumulating the results into the corresponding α vectors. The remaining arithmetic and dataflow are unchanged.

$\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit The polynomial arithmetic in `ComputePoly` repeatedly performs multiplications between base-field elements and extension-field elements, i.e. MAC$_0$, MAC$_2$, MAC$_4$ in Figure 4. For the base fields $\mathbb{F}_2$ and $\mathbb{F}_{2^4}$, the base-field elements need to be lifted (brought) to the extension field before multiplication can take place. A naive way to perform this multiplication is to first apply the lifting operation, and then separately perform generic $\mathbb{F}_{2^8}$-by-$\mathbb{F}_{2^8}$ multiplication.

Previous work on MPCitH schemes shows that fusing the lift and multiply operations can lead to substantial area improvements: lifting an element of $\mathbb{F}_2$ to $\mathbb{F}_{2^8}$ amounts to zero-padding, so the fused operation reduces to selecting either zero or the $\mathbb{F}_{2^8}$ operand [MNKV26]. In this work, we show that this idea can be extended to the $\mathbb{F}_{2^4}$ base field to design an area-efficient *fused* $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit. For the $\mathbb{F}_{2^4}$ base field, the lift to $\mathbb{F}_{2^8}$ is performed through a non-trivial field morphism where the generator $\rho \in \mathbb{F}_{2^4}$ is mapped to $\xi^7 + \xi^6 + \xi^5 \in \mathbb{F}_{2^8}$ [BBFR25]. As the morphism is fixed and linear, the lifted value of any input $a \in \mathbb{F}_{2^4}$ can be expressed as a linear combination of four fixed elements in $\mathbb{F}_{2^8}$.

Figure 5 shows our proposed $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply structure. We start by multiplying the input $b \in \mathbb{F}_{2^8}$ with the four lifted-basis elements (constants $\in \mathbb{F}_{2^8}$) to generate four partial products. In hardware, multiplying an arbitrary field element (i.e. b) by a constant is much cheaper than generic multiplication between two arbitrary field elements. Moreover, the first lifted-basis element is 1, so its partial product is obtained by directly forwarding b. Four 8-bit AND gates select a given partial product only if the corresponding bit of a is 1. The final stage combines (XORs) the four selected partial products to produce the final result.

Table 2 compares our fused $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit with a generic $\mathbb{F}_{2^8}$ multiplier and a separate lift-and-multiply design. Our proposed unit reduces the LUT cost of *separate* lift and multiply operations from 41 to 26 LUTs. Since our $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit is used to replace multiple costly instances of $\mathbb{F}_{2^8}$-by-$\mathbb{F}_{2^8}$ multipliers, the total area footprint of our `ComputePoly` module is reduced significantly.

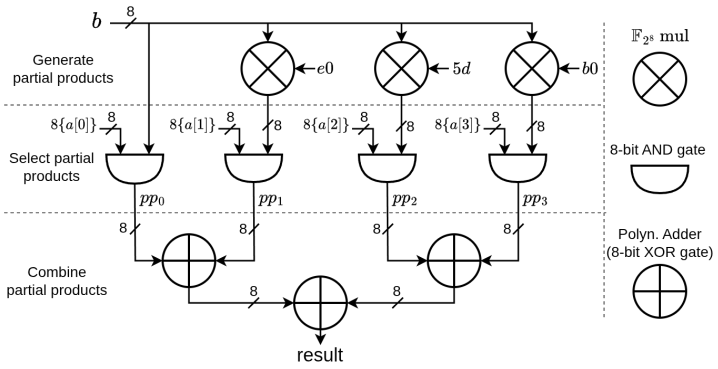

Figure 5: Structure of the proposed fused $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit.

Table 2: Area comparison of different multiplication circuits.

Design	Description	LUTs
$\mathbb{F}_{2^8}$ multiplier	Baseline $\mathbb{F}_{2^8}$ multiplier	39
Separate lift + multiply	$\mathbb{F}_{2^4}$ lift followed by generic $\mathbb{F}_{2^8}$ multiplication	41
Proposed $\mathbb{F}_{2^4}$-to-$\mathbb{F}_{2^8}$ lift-and-multiply unit	Optimized fused implementation (Fig. 5)	26

6 Operation Scheduling

To achieve maximal performance on our hardware design, a carefully optimized operation schedule is required. Maximizing the utilization of the main latency-contributing components, such as the `Enc` PRG, is critical to achieving high performance. We provide detailed scheduling diagrams for the MQOM-L1-gf16-fast-r3 instance, but the discussion and insights directly generalize to all of our proposed designs and parameter sets.

6.1 Key Generation

During key generation, the SHAKE engine first expands `seed_key` to derive the secret solution x and the seed `mseed_eq`, which it uses to generate the `seed_eq` seeds used to derive the packed MQ equations. Once the first four `seed_eq` values are available, the `Enc` PRG starts expanding the packed MQ equations while SHAKE continues producing the remaining seeds. `MatSeq2Triang` reformats the resulting PRG output into the $\hat{A}_i$, $\hat{b}_i$ coefficient stream consumed immediately by `ComputePoly`, which computes the public vector $\hat{y}$. The vector $\hat{y}$ is packed with `mseed_eq` to obtain the public key `pk`, which is then packed with the secret vector x to obtain the secret key `sk`.

6.2 Signature Generation

During signing, the SHAKE engine starts by hashing the message. Concurrently, the `Enc` PRG begins `BLC.Commit`. First, it derives the root seeds for all τ protocol repetitions. Then, one repetition at a time, it expands the GGM trees, commits to the leaf seeds, and generates

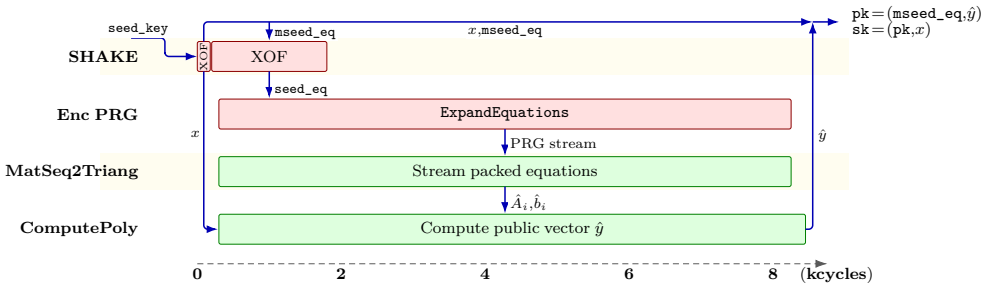

Figure 6: Key-generation schedule for MQOM-L1-gf16-fast-3r

the PRG stream used by `BLC.FoldingComp` to compute x_0, u_0, u_1, and Δ_x. Simultaneously, the SHAKE engine hashes the generated commitments to compute the `hash_ls_com`$[e]$ values on the fly. Once `BLC.Commit` finishes, the SHAKE engine generates the `seed_eq` seeds, and the `Enc` PRG starts expanding the packed MQ equations. The resulting $\hat{A}_i$,$\hat{b}_i$ coefficient stream is consumed immediately by `ComputePoly` to obtain α_0 and α_1. Concurrently, the `hash_ls_com` values are hashed to generate $\texttt{com}_1$. Once the α_0 and α_1 vectors become available, the SHAKE engine computes $\texttt{com}_2$, allowing the final verifier challenge to be derived.

The SHAKE engine then performs the grinding procedure, which produces the hidden-leaf indices $i^*[e]$ for all $e \in [0,\tau-1]$. Finally, `GetSiblingPath` derives the sibling-path indices, after which the `Enc` PRG recomputes the required sibling nodes and hidden-leaf commitments to produce the opening included in the signature.

The only scheduling difference for the 5-round variants is that $\texttt{com}_1$ must be computed before expanding `mseed_eq`, since it is used to derive the matrix Γ, which is required to compute α_0 and α_1 on the fly.

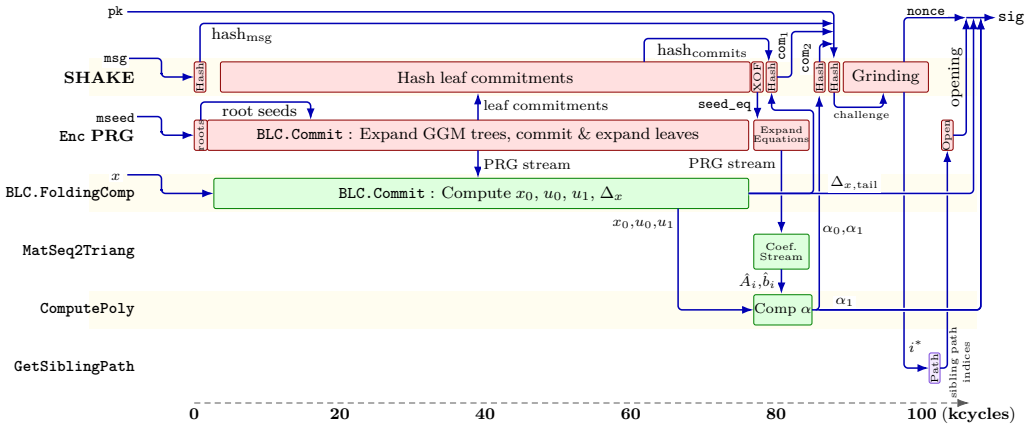

Figure 7: Signing schedule for MQOM-L1-gf16-fast-3r

`BLC.Commit` Latency To maximize hardware utilization during the computationally expensive `BLC.Commit` subroutine (Fig. 7), we tailor our datapath so that the `Enc` PRG, SHAKE engine, and `BLC.FoldingComp` module are closely matched in throughput. The `Enc` PRG generally determines the completion time of this phase and thus lies on the scheduling critical path, although the SHAKE engine can be the limiting factor for parameter sets that require relatively fewer calls to the `Enc` PRG for leaf expansion. Since `BLC.Commit` and `BLC.Eval` perform closely related computations, the same considerations apply to `BLC.Eval` during verification (Fig. 8).

6.3 Signature Verification

During signature verification, the SHAKE engine first hashes the message and derives the verifier challenge from the public key and the commitment values included in the signature. The resulting hidden-leaf indices $i^*[e]$ are supplied to `GetSiblingPath`, which derives the sibling-path indices required by `BLC.Eval` and the corresponding evaluation points $r_{\text{eval}}[e]$. The `Enc` PRG then performs `BLC.Eval`: it reconstructs the GGM trees from the opening included in the signature, recomputes the leaf commitments, and expands the reconstructed leaf seeds. The generated PRG stream is consumed by `BLC.FoldingComp`, together with $\Delta_{x,\text{tail}}$ and r_{eval}, to compute x_{eval} and u_{eval}. Concurrently, the SHAKE engine hashes the reconstructed leaf commitments to recompute the commitment digests `hash_ls_com`$[e]$ and derive $\texttt{com}'_1$.

Once `BLC.Eval` finishes, the SHAKE engine generates the `seed_eq` seeds, and the `Enc` PRG starts expanding the packed MQ equations. The resulting $\hat{A}_i, \hat{b}_i$ coefficient stream is processed immediately by `ComputePoly`, which uses x_{eval}, u_{eval}, r_{eval}, α_1, and $\hat{y}$ to recompute α_0. The SHAKE engine then hashes α_0 and α_1 to obtain $\texttt{com}'_2$. The signature is accepted only if `grind_val` $= 0$ and the recomputed $\texttt{com}'_1$ and $\texttt{com}'_2$ match the corresponding values included in the signature.

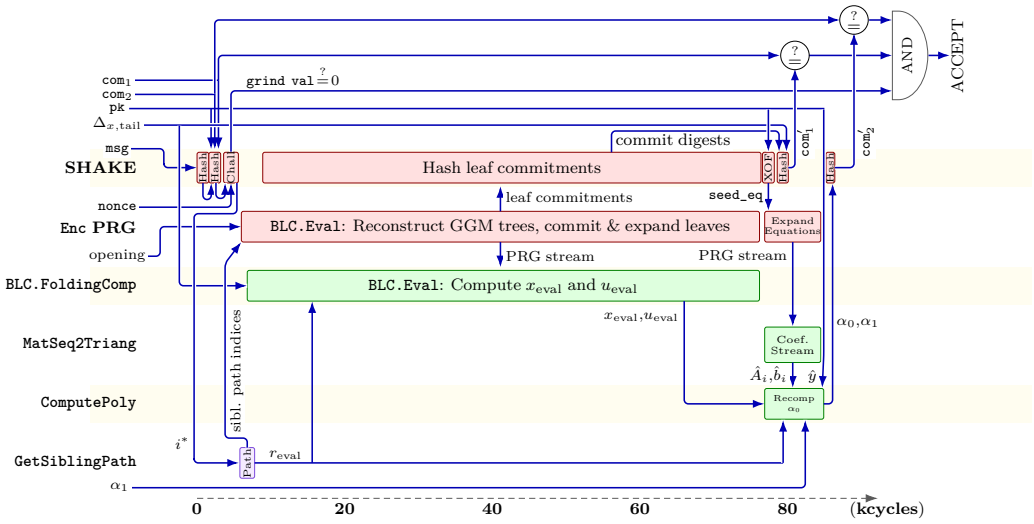

Figure 8: Verification schedule for MQOM-L1-gf16-fast-3r

7 Implementation and Evaluation

We implement the proposed MQOM accelerator in SystemVerilog RTL and synthesize and implement it for the proposed NIST L1 parameter sets using Xilinx Vivado `2024.2`, targeting the Xilinx Artix-7 `XC7A200T-3` FPGA. Cycle counts are obtained from RTL simulation, while resource utilization and maximum operating frequency are obtained from the post-implementation reports. All implemented (top-level) designs have been verified for input/output equivalence against the reference implementation and checked for keygen-sign-verify consistency. Since our polynomial datapath rearranges the internal operations, the intermediate values are generally not identical.

7.1 Impact of MQOM Variants

The computational cost of polynomial computations during `ComputePAlpha` (Alg. 7) and `RecomputePAlpha` is substantial. However, our `ComputePoly` module is able to perform

these computations with very high throughput. Hence, almost all cycles spent on polynomial arithmetic can be overlapped with equation expansion (Fig. 7 & 8). Moreover, `BLC.Commit` and `BLC.Eval`, which account for a large fraction of the total signing and verification latency, require few arithmetic operations. These observations indicate that the `Enc` PRG, rather than the arithmetic datapath, is the primary source of latency in our design for all MQOM instances. Hence, the number of calls to the `Enc` PRG (Table 3) can provide a useful first-order predictor of relative hardware latency between MQOM instances.

Table 3: Number of `Enc` PRG calls for the fast NIST L1 MQOM parameter sets.

MQOM2 Instance	Parameters		Enc PRG Calls				(ratio)
			Enc calls	Expand Eq.	BLC	Total	Expand Eq./
	λ	$\hat{m}=\frac{m}{\mu}$	per $(\hat{A}_i,\hat{b}_i)$	Enc calls	Enc calls	Enc calls	BLC Enc Calls
L1-gf2-fast-3r/5r	128	20	815	16 300	21 726	38 026	0.750
L1-gf16-fast-3r/5r	128	28	104	2 912	26 078/21 726	28 990/24 638	0.112/0.134
L1-gf256-fast-3r/5r	128	48	77	3 696	34 782/26 078	38 478/29 774	0.106/0.142

$\mathbb{F}_2$ vs. $\mathbb{F}_{2^4}$ vs $\mathbb{F}_{2^8}$ We observe that the $\mathbb{F}_{2^4}$ instances require the fewest `Enc` calls and are therefore expected to offer the highest performance. The $\mathbb{F}_2$ instances remain attractive because their base-field arithmetic is particularly inexpensive to implement in hardware. In contrast, the $\mathbb{F}_{2^8}$ instances combine a larger number of `Enc` calls with the most expensive arithmetic datapath, since their base-field operations require general $\mathbb{F}_{2^8}$ multiplication. Our optimizations remain applicable to these instances, but we do not implement them in this work and leave their evaluation for future work.

Since the `Enc` PRG is the main source of latency, we design our initial implementation around keeping all four `Enc` cores active to minimize total latency. All four parallel `Enc` cores are therefore active throughout the `BLC.Commit`, `BLC.Eval`, and `ExpandEquations`. At any point during `ExpandEquations`, we expand a single packed equation (i loop in Alg. 2) per `Enc` core. To perform all computations on-the-fly this requires instantiating the same number of polynomial computation lanes in parallel. A side-effect of this approach is that the circuit area of the `ComputePoly` module increases significantly. This strategy is well suited to the $\mathbb{F}_2$ instances: their arithmetic is the most efficient to implement and `ExpandEquations` accounts for a large fraction of their total `Enc` workload (Table 3).

For the $\mathbb{F}_{2^4}$ instances, however, the ratio of `ExpandEquations` calls to BLC-related calls is considerably smaller. The more expensive `ComputePoly` datapath is active only during the comparatively short equation-expansion phase and remains idle throughout most of `BLC.Commit` and `BLC.Eval`. This leads to significant circuit underutilization, since `ComputePoly` is the largest individual module in our architecture (Table 4). Hence, we also implement Area-Time-Product (ATP)-optimized $\mathbb{F}_{2^4}$ designs that only utilize one or two `Enc` cores during the `ExpandEquations` subroutine in `ComputePAlpha` and `RecomputePAlpha`. This allows us to significantly reduce the area footprint of our `ComputePoly` datapath. Key generation continues to use all four `Enc` cores, since the datapath required to compute the public vector $\hat{y}$ is comparatively inexpensive.

3r vs 5r The 5-round MQOM variants introduce additional implementation complexity. The matrix Γ must be generated, stored, and multiplied with the polynomial coefficients, and its dependence on $\mathtt{com}_1$ imposes an additional scheduling constraint. The MQOM specification similarly observes that the communication savings obtained through the 5-round variant are relatively small, particularly for the $\mathbb{F}_2$ instances, whereas the 3-round variant reduces the implementation complexity [BBFR25]. For hardware, however, the 5-round variant offers an additional advantage: compared to the 3-round variant, it reduces the required number of `Enc` calls during `BLC.Commit` and `BLC.Eval` by up to 16.7% for the $\mathbb{F}_{2^4}$ instances, which suggests improved hardware performance. Hence, we choose to implement both the `3r` and `5r` variants for these instances.

7.2 Resource Utilization

Table 4 presents the detailed resource utilization of our proposed accelerator and its submodules for NIST L1 parameter sets. For the high-throughput gf16-r3-4 design, the arithmetic modules (`BLC.FoldingComp`, `MatSeq2Triang`, and `ComputePoly`) account for 63.7% of the total LUTs and 53.1% of the total FFs. Most of this area belongs to `ComputePoly`, which occupies 58.6% of the total LUTs and 42.3% of the FFs. Reducing the parallelism of the polynomial datapath by a factor of four decreases the `ComputePoly` area utilization by 67.9% in LUTs and 31.0% in FFs. This reduces the total area of our design by 39.8% in LUTs and 13.1% in FFs.

The SHAKE and `Enc` PRG engines together account for a further 33.5% of the LUTs and 42.2% of the FFs. Furthermore, we observe that our streaming and on-the-fly processing optimizations keep the on-chip memory footprint small, as the L1 designs require only 4.5 BRAM tiles (RAMB36K).

Table 4: Detailed utilization of our proposed MQOM hardware accelerator for NIST L1 parameters (gf2-r3-4 / gf16-r3-4 / gf16-r3-1 / gf16-r5-4 / gf16-r5-1). The final number denotes the number of parallel `ComputePoly` lanes used during `ComputePAlpha` and `RecomputePAlpha`.

Module	LUT	FF	BRAM
Top-Level FSM	674 / 673 / 658 / 705 / 701	560 / 567 / 565 / 562 / 563	0 / 0 / 0 / 0 / 0
SHAKE Engine	3,479 / 3,512 / 3,508 / 3,484 / 3,513	2,112 / 2,085 / 2,094 / 2,091 / 2,084	0.5 / 0.5 / 0.5 / 0.5 / 0.5
`Enc` PRG Engine	5,197 / 5,232 / 5,251 / 5,260 / 5,248	3,399 / 3,382 / 3,376 / 3,376 / 3,399	0 / 0 / 0 / 0 / 0
`BLC.FoldingComp`	688 / 993 / 993 / 981 / 980	1,197 / 759 / 759 / 756 / 756	0 / 0 / 0 / 0 / 0
`MatSeq2Triang`	341 / 322 / 321 / 331 / 319	642 / 638 / 637 / 640 / 637	0 / 0 / 0 / 0 / 0
`ComputePoly`	13,546 / 15,289 / 4,915 / 15,099 / 4,992	5,435 / 5,485 / 3,782 / 4,440 / 1,893	0 / 0 / 0 / 0 / 0
`GetSiblingPath`	54 / 56 / 58 / 58 / 59	52 / 52 / 52 / 52 / 52	0 / 0 / 0 / 0 / 0
Main Memory	0 / 0 / 0 / 0 / 0	0 / 0 / 0 / 0 / 0	4 / 4 / 4 / 4 / 4
Total	**23,979 / 26,077 / 15,704 / 25,918 / 15,812**	**13,397 / 12,968 / 11,265 / 11,917 / 9,384**	**4.5 / 4.5 / 4.5 / 4.5 / 4.5**

Scaling to L3/L5 We consider extending the architecture while keeping the same number of parallel encryption cores and polynomial computation lanes. For L3 and L5, MQOM uses Rijndael-256-256 instead of AES-128. This doubles the block width, the number of S-boxes in the encryption round datapath, and the width of the input/output logic. With our one-round-per-cycle architecture, the block-encryption latency increases from 11 to 15 cycles, a factor of approximately 1.36×.

The polynomial datapath also grows with the protocol parameters. In particular, τ increases from 17 at L1 to 27 and 36 at L3 and L5, respectively. For a fixed number of computation lanes, the arithmetic replicated across repetitions therefore grows by approximately 1.59× and 2.12×. These factors are comparable to the increase in the encryption datapath, suggesting that their relative area contributions remain broadly similar.

Latency increases through both the longer block-cipher evaluations and the larger protocol parameters. Increasing m enlarges both the number and size of the MQ equations, extending coefficient generation and polynomial computation. The datapaths remain matched during equation streaming, so the additional coefficients are still processed as they are generated. Increasing τ also requires more GGM trees to be processed, which further increases L3/L5 signing and verification latency.

7.3 Performance

Table 5 reports the performance of our accelerator and compares it with optimized MQOM software and related PQC signature hardware implementations. All of our designs achieve a maximum operating frequency of 260 MHz. The stochastic grinding procedure introduces runtime variability to the signing procedure. We report the median signing latency for all of our implemented instances. Key generation and verification are both constant-time. The fastest design

(L1-gf16-r5-4) completes `KeyGen` in 0.0325 ms, `Sign` in a median of 0.372 ms, and `Verify` in 0.286 ms. Compared with L1-gf16-r3-4, the 5-round design reduces the signing and verification latency by 6.9% and 12.7%, respectively, while requiring approximately the same area.

Table 5: Comparison of our proposed MQOM hardware designs with optimized MQOM software and related PQC DSA hardware implementations.

Scheme	Parameter Set	Resources LUT/FF/DSP/BRAM	Area[a]	Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	Sign ATP Area×s	Verify ATP Area×s	Comb. ATP[a] Area×s
MQOM (This work)	L1-gf2-fast-r3-4	23,979/13,397/0/4.5	25,329	260	0.0453/0.1742	0.1298/0.4992	0.1099/0.4229	12.6	10.7	27.8
	L1-gf16-fast-r3-4	26,077/12,968/0/4.5	27,427	260	0.0084/0.0325	0.1038/0.3992	0.0851/0.3271	10.9	9.0	20.8
	L1-gf16-fast-r3-2	19,578/11,825/0/4.5	20,928	260	0.0084/0.0325	0.1117/0.4296	0.0930/0.3575	9.0	7.5	17.2
	L1-gf16-fast-r3-1	15,704/11,265/0/4.5	17,054	260	0.0084/0.0325	0.1277/0.4911	0.1090/0.4191	8.4	7.1	16.1
	L1-gf16-fast-r5-4	25,918/11,917/0/4.5	27,268	260	0.0084/0.0325	0.0966/0.3716	0.0743/0.2856	10.1	7.8	18.8
	L1-gf16-fast-r5-2	18,912/10,168/0/4.5	20,262	260	0.0084/0.0325	0.1047/0.4025	0.0823/0.3165	8.2	6.4	15.2
	L1-gf16-fast-r5-1	15,812/9,384/0/4.5	17,162	260	0.0084/0.0325	0.1202/0.4622	0.0981/0.3772	7.9	6.5	15.0
MQOM SW[b] (AVX-512+GFNI) [BBFR25]	L1-gf2-fast-r3	–/–/–/–	–	2,500	0.86/0.344	2.50/1.000	2.11/0.844	–	–	–
	L1-gf16-fast-r3	–/–/–/–	–	2,500	0.24/0.096	1.59/0.636	1.24/0.496	–	–	–
	L1-gf16-fast-r5	–/–/–/–	–	2,500	0.25/0.100	1.53/0.612	1.18/0.472	–	–	–
MQOM SW (Cortex-M4) [BBFR25]	L1-gf2-fast-r3	–/–/–/–	–	16	24.18/1,511.25	118.67/7,416.88	112.50/7,031.25	–	–	–
	L1-gf16-fast-r3	–/–/–/–	–	16	5.90/368.75	64.96/4,060.00	53.29/3,330.63	–	–	–
	L1-gf16-fast-r5	–/–/–/–	–	16	5.90/368.75	66.76/4,172.50	48.24/3,015.00	–	–	–
MQOM SW+HW-AES-128 [BBFR25]	L1-gf2-fast-r3	–/–/–/–	–	180	14.38/79.89	155.73/865.17	147.42/819.00	–	–	–
	L1-gf16-fast-r3	–/–/–/–	–	180	6.64/36.89	66.00/366.67	51.64/286.89	–	–	–
	L1-gf16-fast-r5	–/–/–/–	–	180	6.64/36.89	70.74/393.00	49.44/274.67	–	–	–
ML-DSA [NKV25]	L2	54,942/29,309/16/29.0	65,242	230	0.0049/0.0213	0.024/0.1043	0.0066/0.0287	6.8	1.9	10.07
Falcon[c] [OZZ+25]	L1	79,065/46,389/226/45.0	115,165	65	–/–	0.16/2.46	–/–	283.3	–	–
SPHINCS+ [ALCZ20]	128s	48,231/72,514/0/11.5	51,681	250/500	–/–	–/12.40	–/0.07	640.8	3.6	644.5
	128f	47,991/72,505/1/11.5	51,541	250/500	–/–	–/1.01	–/0.16	52.1	8.2	60.3
CROSS [KAB+25]	RSDP-1-f	26,741/11,678/0/45.0	40,241	118	–/0.035	–/0.478	–/0.409	19.2	16.5	37.1
LESS [BWMG23]	L1-b	54,800/39,900/0/59.5	72,650	200	0.029/0.14	5.20/26.02	5.156/25.78	1,890.4	1,872.9	3,773.4
	L1-i	54,800/39,900/0/59.5	72,650	200	0.077/0.38	5.13/25.63	5.093/25.47	1,862.0	1,850.4	3,740
	L1-s	54,800/39,900/0/59.5	72,650	200	0.174/0.87	4.17/20.83	4.137/20.69	1,513.3	1,503.1	3,079.6
MAYO [HSK+24]	L1-f	106,000/38,000/2/45.5	119,850	75	0.0058/0.08	0.031/0.43	0.0046/0.05	51.5	6.0	67.1
UOV [BCD+25]	Ip-classic	33,221/24,097/2/108.5	65,971	93.5	3.5/37.433	0.0075/0.08	0.0064/0.068	5.3	4.5	2,479.3
	Ip-pkc	32,134/22,969/2/81.0	56,634	91.4	4.2/45.952	0.0075/0.082	0.192/2.10	4.6	118.9	2,726.0
	Ip-pkc+skc	32,422/23,262/2/48.0	47,022	94.8	3.8/40.084	0.353/3.724	0.192/2.025	175.1	95.2	2,155.2
	Is-classic	44,974/27,433/2/140.0	87,174	92.2	9.9/107.38	0.013/0.141	0.013/0.141	12.3	12.3	9,385.3
	Is-pkc	29,385/25,328/2/110.0	62,585	94.8	11.9/125.53	0.013/0.137	0.284/2.996	8.6	187.5	8,052.4
	Is-pkc+skc	28,947/24,444/2/66.0	48,947	90.8	11.1/122.25	0.84/9.251	0.284/3.128	452.8	153.1	6,589.7
Mirath [MNKV26]	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.500/1.969	61.1	55.1	118.0
SDitH-v2[c] [HHH+26]	L1-s (HP1)	24,442/9,456/1/3.0	25,442	130.0	–/–	9.89/76.1	–/–	1,936.1	–	–
SDitH-v1 [DHSY24]	L1-GF256	16,592/8,778/0/164.5	65,942	164	0.055/0.34	6.73/41.04	8.688/52.98	2,706.3	3,493.6	6,222.3
	L1-GF251	17,423/16,336/196/164.5	86,373	164	0.057/0.35	21.45/130.77	23.403/142.71	11,295.0	12,326.3	23,651.5
PICNIC [KRR+20]	L1	90,337/23,105/0/52.5	106,087	125	–/–	0.03/0.25	0.030/0.24	26.5	25.5	52.0

[a] Area-Time-Product (ATP) is area times runtime in seconds, with area measured as #LUTs+100×#DSPs+300×#BRAMs [YCH22].
Combined ATP includes Sign and Verify, plus KeyGen where reported.
[b] AMD Ryzen Threadripper PRO 7995WX @ 2.5GHz, GCC 15.2.0 (turbo boost OFF).
[c] Only supports `Sign`.

The ATP-optimized, lower-throughput designs substantially reduce area while introducing only a modest latency increase. Compared to L1-gf16-r5-4, the L1-gf16-r5-1 design reduces combined area by 37.1% and achieves the lowest ATP of 15.0, at the cost of a 26.4% increase in total latency.

Comparison with Optimized MQOM Software Implementations Each speed-up is computed as the ratio of software to hardware runtime. The combined speed-up is the ratio of the sums of the KeyGen, Sign, and Verify runtimes.

Compared to the best-performing AVX-512+GFNI[4] SW implementation of MQOM [BBFR25] (@ 2.5 GHz), our fastest L1-gf16-fast-r5-4 design (@ 260 MHz) achieves 3.08×, 1.65×, and 1.65× speed-ups for KeyGen, Sign, and Verify, respectively, with a combined speed-up of 1.72×. Our L1-gf2-fast-r3-4 design achieves corresponding speed-ups of 1.97×, 2.00×, and 2.00×, with a combined speed-up of 2.00×.

Compared to the available constant-time SW implementation targeting the Cortex-M4 microcontroller unit (MCU) (@ 16 MHz), our high-performance L1-gf2-fast-r3-4 design achieves 8,675×, 14,858×, and 16,626× speed-ups for KeyGen, Sign, and Verify, respectively, with a combined speed-up of 14,557×. Our L1-gf16-fast-r5-4 design achieves corresponding speed-ups of 11,346×, 11,228×, and 10,557×, with a combined speed-up of 10,956×. Our

[4]The Galois Field New Instructions (GFNI) instruction set brings dedicated $\mathbb{F}_{2^8}$ multiplication to x86 platforms.

ATP-optimized L1-gf16-fast-r5-1 design achieves corresponding speed-ups of 11,346×, 9,027×, and 7,993×, with a combined speed-up of 8,666×, while being significantly more area efficient than the proposed high-performance designs.

Compared to the implementation targeting a Cortex-M4 MCU (STM32F437 [STM] @ 180 MHz) that features dedicated hardware AES-128 acceleration, our L1-gf2-fast-r3-4 design achieves 459×, 1,733×, and 1,937× speed-ups for KeyGen, Sign, and Verify, respectively, with a combined speed-up of 1,609×. Our L1-gf16-fast-r5-4 design achieves corresponding speed-ups of 1,135×, 1,058×, and 962×, with a combined speed-up of 1,022×. The ATP-optimized L1-gf16-fast-r5-1 design achieves corresponding speed-ups of 1,135×, 850×, and 728×, with a combined speed-up of 808×.

Comparison with Related Hardware Works Direct comparisons between different signature schemes must be interpreted with care because of differences in algorithms, supported operations, FPGA target devices, and optimization objectives. To make the comparisons more fair, we only include works that target the same resource-constrained Artix-7 FPGA family as we do. We also focus the comparisons on unified KeyGen-Sign-Verify implementations, but also include some interesting designs that only implement signature generation, since unified hardware designs do not exist for some post-quantum DSAs. All of the presented peer works, except the area-optimized SDitH-v2 design by Halmans et al. [HHH+26], are either ATP-optimized or performance optimized, which makes close comparisons more feasible. Since comparing each one of our designs with every single peer design is infeasible, we choose to only compare the best ATP design from every peer work with our own ATP-optimized L1-gf16-r5-1 design.

Comparing to the lattice-based PQC DSA standards ML-DSA [Nat24a] and FN-DSA [FHK+22] is difficult due to the inherent differences between lattice-based and MPCitH-based schemes. The unified ML-DSA design by Norga et al. [NKV25] is faster than our MQOM design. However, since our design is 3.8× smaller, our combined ATP is larger only by a factor of 1.5×. Compared to the Falcon (only implementing `Sign`) by Ouyang et al. [OZZ+25], our design performs signature generation 5.32× faster, while our area is also reduced by a factor of 6.71×.

Compared to the SPHINCS+ 128f design by Amiet et al. [ALCZ20], our design performs signature generation 2.19× faster, while verification is 2.36× slower. Our MQOM design is 3.0× more area efficient, which makes our combined ATP metric 4.02× lower.

Compared to the available MPCitH-based HW implementations, our ATP-optimized MQOM design is the most area-efficient, while being slower only when compared to the PICNIC design [KRR+20]. Compared to Mirath [MNKV26], our ATP is improved over a factor of 7.87×. Compared to the sign-only area-optimized SDitH-v2 design, our signing latency is lower by a factor of 164.65×, while our area is also lower by 1.48×. Compared to the older unified SDitH-v1 design, our combined ATP is improved over a factor of 414.8×, while compared to PICNIC design our ATP is 3.47× better.

Next, we compare our design with state-of-the-art HW implementations of multivariate schemes. Compared to the UOV-Ip-pkc+skc implementation, which has the lowest combined ATP among the UOV rows in Table 5, our total Keygen-Sign-Verify latency is lower by a factor of 52.57×, while our area is reduced by 2.74× and our combined ATP is lower by 143.68×. This aggregate comparison is strongly influenced by UOV's expensive Keygen, whereas our design spends most of its runtime in Sign and Verify. In particular, UOV-Ip-classic has shorter Sign and Verify latencies than our design despite its higher combined ATP, which could be beneficial in a scenario where the same key is reused multiple times. Specifically, its Sign ATP and Verify ATP are 5.3 and 4.5, respectively, compared with 7.9 and 6.5 for our design. Compared to the MAYO L1-f implementation, our total latency is increased by a factor of 1.56×. However, our combined ATP is 4.47× better.

We further compare with available hardware implementations of code-based signatures CROSS and LESS. Our MQOM design performs similarly to cross, being only 1.06× faster.

However, due to our better area-efficiency, our combined ATP is 2.47× better. Compared to LESS-L1-s, our ATP-optimized MQOM implementation is 48.62× faster, while using 4.23× less FPGA area.

Side-Channel Leakage Exposure The runtime of our implementation is independent of secret data and thus our design is protected against timing attacks. However, we have not integrated any countermeasure against power or electromagnetic side-channel attacks. The `Enc` PRG processes secret seeds as plaintext. Therefore, intermediate states of the `Enc` datapath may leak secret-related information through power or electromagnetic side channels. Similarly, polynomial arithmetic processing secret-related data may also leak sensitive information. Therefore, a protected implementation with masking, hiding, or shuffling would need to be integrated into the MQOM implementation to protect against power or electromagnetic side-channel attacks. A prior work has provided a masked implementation of MQOM on a Cortex-M4 platform [HCCY26]. A similar set of overhead factors (area, latency, randomness) is expected to apply the masking countermeasure to our hardware implementations of MQOM. However, determining the concrete overhead factors for integrating first-order masking in our implementation requires further exploration and is left for future work.

8 Conclusion

MQOM is a promising MPCitH-based DSA, which relies primarily on symmetric primitives and arithmetic in Galois fields. In this work we evaluated the performance and suitability of MQOM on hardware platforms, including the use of *correlated GGM trees*. We observe that, especially compared to MPCitH-based schemes that rely on the *one GGM tree* optimization (e.g. Mirath), the design flexibility is significantly increased and hardware cost is significantly reduced. Our design, in which data generated by the optimized symmetric primitive cores is immediately consumed by the parallelized arithmetic datapath on-the-fly, avoids large on-chip memory requirements. Overall, at NIST L1 parameters, our design achieves similar ATP results as state-of-the-art ML-DSA implementations, which has not been achieved by any hardware design of MPCitH-based and other non-lattice-based schemes thus far. Compared to optimized AVX-512+GFNI software implementations of MQOM, our design is up to twice as fast despite a significantly lower clock speed (9.6×) on the Artix-7 FPGA. Finally, we remark that our design can be easily adapted to other MQOM parameter sets and is sufficiently flexible to accommodate potential future changes to the scheme (e.g. NIST Round 3). Our design can also be adapted to create efficient implementations of other PQC signatures relying on cGGM trees.

Acknowledgments

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), the Horizon Europe grant agreement (101308625 EQUIP), and the Cybersecurity Research Program Flanders (CRPF) with reference number VOEWICS02.

References

[AAA+25] Marius A. Aardal, Gora Adj, Diego F. Aranha, Andrea Basso, Isaac Andrés Canales Martínez, Jorge Chávez-Saab, Maria Corte-Real Santos, Pierrick Dartois, Luca De Feo, Max Duparc, Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Krijn Reijnders, Damien Robert, Francisco Rodríguez Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign: Algorithm specifications and supporting documentation, July 2025. Version 2.0.1, Round 2 submission to the NIST Additional Digital Signature Schemes project.

[ABC+26] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, Quynh Dang, Thinh Hung Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. Status report on the second round of the additional digital signature schemes for the NIST post-quantum cryptography standardization process. NIST Internal Report NIST IR 8610, National Institute of Standards and Technology, Gaithersburg, MD, May 2026.

[ALCZ20] Dorian Amiet, Lukas Leuenberger, Andreas Curiger, and Paul Zbinden. Fpga-based sphincs$^+$ implementations: Mind the glitch. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*, pages 229–237. IEEE, 2020.

[BBB+25] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST v2 Algorithm Specifications. Technical report, National Institute of Standards and Technology, February 2025. Round 2 submission.

[BBD+25] Joppe W. Bos, Olivier Bronchain, Léo Ducas, Serge Fehr, Yu-Hsuan Huang, Thomas Pornin, Eamonn W. Postlethwaite, Thomas Prest, Ludo N. Pulles, and Wessel van Woerden. HAWK: Lattice Isomorphism-based Signature Scheme – Round 2 Specification. Technical report, National Institute of Standards and Technology, May 2025. Round 2 submission.

[BBdSG+23] Carsten Baum, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Emmanuela Orsini, Lawrence Roy, and Peter Scholl. Publicly verifiable zero-knowledge and post-quantum signatures from vole-in-the-head. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 581–615, Cham, 2023. Springer Nature Switzerland.

[BBFR25] Ryad Benadjila, Charles Bouillaguet, Thibauld Feneuil, and Matthieu Rivain. MQOM: MQ on my mind. algorithm specifications and supporting documentation (version 2.1). Technical report, CryptoExperts and Sorbonne Université, September 2025. Released September 22, 2025.

[BBM+24] Carsten Baum, Ward Beullens, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. One tree to rule them all: Optimizing GGM trees and owfs for post-quantum signatures. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December*

9-13, 2024, Proceedings, Part I, volume 15484 of *Lecture Notes in Computer Science*, pages 463–493. Springer, 2024.

[BCC+25] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO specification document: Round 2 version. Round 2 submission to the NIST Additional Post-Quantum Cryptography Digital Signature Standardization Process, February 2025.

[BCD+25] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. Uov: Unbalanced oil and vinegar (v2.0), February 2025. Available at https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/uov-spec-round2-web.pdf.

[BF26] Ryad Benadjila and Thibauld Feneuil. Breaking the myth of MPCitH inefficiency: Optimizing MQOM for embedded platforms. Cryptology ePrint Archive, Paper 2026/078, 2026.

[BWMG23] Luke Beckwith, Robert Wallace, Kamyar Mohajerani, and Kris Gaj. A high-performance hardware implementation of the LESS digital signature scheme. In Thomas Johansson and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 14th International Workshop, PQCrypto 2023, College Park, MD, USA, August 16-18, 2023, Proceedings*, volume 14154 of *Lecture Notes in Computer Science*, pages 57–90. Springer, 2023.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography, 2016.

[DHSY24] Sanjay Deshpande, James Howe, Jakub Szefer, and Dongze Yue. Sdith in hardware. *IACR Cryptol. ePrint Arch.*, page 69, 2024.

[Fen24] Thibauld Feneuil. The polynomial-iop vision of the latest mpcith frameworks for signature schemes. ACCESS Seminar, online, October 2024. Slides.

[FHK+22] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. Technical report, falcon-sign.info, 2022. Supporting documentation.

[FIH+25] Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Tsuyoshi Takagi, Haruhisa Kosuge, Kimihiro Yamakoshi, Rika Akiyama, Satoshi Nakamura, Shingo Orihara, and Koha Kinjo. QR-UOV: Specification document. Technical report, Nippon Telegraph and Telephone Corporation, February 2025. Round 2 submission to the NIST Additional Digital Signature Schemes project.

[FR23] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of mpc-in-the-head. In *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part I*, page 441–473, Berlin, Heidelberg, 2023. Springer-Verlag.

[FR25] Thibauld Feneuil and Matthieu Rivain. Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. *J. Cryptol.*, 38(3), June 2025.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.

[GYW+23] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Correction to: Half-tree: Halving the cost of tree expansion in cot and dpf. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages C1–C1, Cham, 2023. Springer Nature Switzerland.

[HCCY26] Yi-Lin Hung, Jiun-Peng Chen, Ho-Lin Chen, and Bo-Yin Yang. High-Order Masking for MQOM v2.1 Signing. Cryptology ePrint Archive, Paper 2026/769, 2026.

[HHH+26] Stef Halmans, Niklas Höher, Dina Hesse, Sanjay Deshpande, Jakub Szefer, and Tim Güneysu. Storing less in-the-head: An area-efficient hardware architecture for SDitH-v2. Cryptology ePrint Archive, Paper 2026/938, 2026.

[HSK+24] Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the multivariate-based MAYO signature scheme using hardware platforms. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3421–3435. ACM, 2024.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

[KAB+25] Patrick Karl, Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Georg Sigl. High-performance FPGA accelerator for the post-quantum signature scheme CROSS. *IACR Cryptol. ePrint Arch.*, page 1161, 2025.

[KRR+20] Daniel Kales, Sebastian Ramacher, Christian Rechberger, Roman Walch, and Mario Werner. Efficient FPGA implementations of lowmc and picnic. In Stanislaw Jarecki, editor, *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, volume 12006 of *Lecture Notes in Computer Science*, pages 417–441. Springer, 2020.

[MBB+25] Carlos Aguilar Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. SD-in-the-Head: Syndrome-Decoding-in-the-Head Signature Scheme – Round 2 Specification. Technical report, National Institute of Standards and Technology, February 2025. Round 2 submission, Version 2.0 (2025-02-05).

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[MNKV26] Stelios Manasidis, Quinten Norga, Suparna Kundu, and Ingrid Verbauwhede. Mpspeed: Implementing and optimizing mpc-in-the-head digital signatures in hardware: Accelerating mirath on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):982–1006, Jul. 2026.

[Nat23] National Institute of Standards and Technology (NIST). Call for additional digital signature schemes for the post-quantum cryptography standardization process. https://csrc.nist.gov/Projects/post-quantum-cryptography/additional-signatures-2023, September 2023.

[Nat24a] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard, 2024.

[Nat24b] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard, 2024.

[NKV25] Quinten Norga, Suparna Kundu, and Ingrid Verbauwhede. ML-DSA-OSH: an efficient, open-source hardware implementation of ML-DSA. *IACR Cryptol. ePrint Arch.*, page 2337, 2025.

[OZZ+25] Yi Ouyang, Yihong Zhu, Wenping Zhu, Bohan Yang, Zirui Zhang, Hanning Wang, Qichao Tao, Min Zhu, Shaojun Wei, and Leibo Liu. Falconsign: An efficient and high-throughput hardware architecture for falcon signature generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):203–226, 2025.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[STM] STMicroelectronics. STM32F437ZG: High-performance arm cortex-m4 mcu with hardware cryptographic acceleration. https://www.st.com/en/microcontrollers-microprocessors/stm32f437zg.html. Accessed: 2026-07-15.

[WCD+25] Lih-Chung Wang, Chun-Yen Chou, Jintai Ding, Yen-Liang Kuan, Jan Adriaan Leegwater, Ming-Siou Li, Bo-Shu Tseng, Po-En Tseng, and Chia-Chun Wang. SNOVA: Proposal for NISTPQC additional digital signature schemes, January 2025. Version 2.0, Round 2 submission to the NIST Additional Digital Signature Schemes project.

[YCH22] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. Pipentt: A pipelined number theoretic transform architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(10):4068–4072, 2022.