IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–25. DOI:XXXXXXXX

MPSpeed: Implementing and Optimizing MPC-in-the-Head Digital Signatures in Hardware

Accelerating Mirath on FPGA

Stelios Manasidis, Quinten Norga, Suparna Kundu and Ingrid Verbauwhede

COSIC, KU Leuven, Leuven, Belgium
stelios443@gmail.com
{quinten.norga,suparna.kundu,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. The Multi-Party Computation (MPC)-in-the-Head (MPCitH) framework enables the construction of post-quantum Digital Signature Algorithms (DSAs), offering competitive public key sizes. However, this comes at a cost of high computational complexity, resulting in high signature generation and verification times.
In this work, we propose a compact and efficient hardware accelerator for `Mirath`, an MPCitH-based DSA and candidate in the ongoing NIST PQC standardization effort. We propose a series of algorithmic and hardware-level optimizations, focusing on `Mirath`'s most critical operations: GGM tree-based polynomial commitments and MPC arithmetic. Firstly, we observe `Mirath` greatly relies on symmetric primitives (SHA3 & AES) during the GGM tree expansion and typically requires a large amount of memory to store the derived tree nodes. We propose an on-the-fly scheduling for generating and computing the GGM tree, such that a minimal amount of GGM tree nodes are stored in memory and their computations can be performed in parallel. Our methodology enables temporarily storing a minimal (and configurable) set of parent nodes in local buffers, from which the low-level tree nodes can be efficiently derived instead of repeatedly doing so from the root seed. This is achieved through a novel, hardware-friendly tree node indexing scheme, which enables efficient traversal through GGM tree nodes using only left and right shifts to find their closest previously computed ancestor. Secondly, we analyze the MPC arithmetic in `Mirath` and propose massively parallel and yet area-efficient arithmetic units, capable of exploiting algorithm-level parallelism in the MPCitH operations. This is achieved by analyzing `Mirath`'s proposed parameter sets and identifying the most hardware-friendly parameters, for which we design highly fine-tuned modules. Finally, we implement our unified design, which supports all `Mirath` operations, on an Artix-7 FPGA and compare its performance against `Mirath`'s AVX2 optimized implementation and state-of-the-art PQC DSA hardware implementations. Compared to an implementation of the MPCitH-based SDitH scheme (TCHES 2024), we reduce on-chip BRAM by up to 81.6% and improve the area-time-product by a factor of 52.7× to 64.8×. Overall, we demonstrate that modern MPCitH constructions can be significantly accelerated in hardware through a combination of algorithmic, architectural and low-level hardware optimizations, in line with real-world performance requirements.

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

1 Introduction

A large quantum computer capable of executing Shor's algorithm [Sho94] poses significant threats to actively deployed public key cryptosystems that use RSA [RSA78] or ECC [Mil85].

Licensed under Creative Commons License CC-BY 4.0.

In anticipation of this threat, the National Institute of Standards and Technology (NIST) organized its first post-quantum cryptography standardization process in 2016 [Nat16], which resulted in the selection of three post-quantum Digital Signature Algorithm (DSA) standards in 2022: ML-DSA [Nat24a], FN-DSA [FHK+22] and SLH-DSA [Nat24b]. Currently, NIST is conducting an additional standardization process [NIS23], with the aim of diversifying the portfolio of PQC DSA standards and providing greater robustness to prepare for unfortunate situations, such as the weakening of the hardness assumptions of one or more already standardized schemes. As of January 2026, the ongoing NIST standardization process comprises 14 candidates, many of which offer low bandwidth costs due to small public keys and/or signatures, at the cost of higher computational complexity [NIS24]. More specifically, there are five candidate schemes belonging to the Multi-Party Computation (MPC)-in-the-Head (MPCitH) family, benefiting from relatively small public keys, including: `Mirath` [AAB+25b], `MQOM` [BBFR25], `PERK` [ABB+25a], `RYDE` [ABB+25b] and `SDitH` [MBB+25]. Compared to the first set of PQC standards, which are mainly lattice-based, the performance of the additional candidates is largely worse and even impractical in some cases.

The MPCitH framework was originally proposed by Ishai et al. [IKOS07], which enables a prover to prove the knowledge of the secret key to a verifier without revealing any additional information. Instead of several live parties running an MPC protocol, a single prover locally emulates an N party MPC protocol *in their head* and computes the commitments of all N parties. A verifier checks the consistency of a subset of the N party computations (typically all-but-one), before deciding to accept or reject the party commitments as authentic. As the consistency of not all N party shares is checked, there is a probability of a corrupted computation being accepted as authentic (false positive), and thus τ iterations of the MPC protocol are performed to increase the soundness of the proof.

In the first generation MPCitH-based schemes, such as Picnic [CDG+17], the simulation of a large number of (additively shared) virtual parties incurs a high computational cost. More recently, the optimized Threshold Computation-in-the-Head (TCitH) framework [FR23, FR25] has been proposed, on which `Mirath` (v2.1) is based: instead of relying on additive secret sharing, TCitH uses *threshold secret sharing* to structure the simulated MPC views so that the emulation cost depends only on a small reconstruction threshold rather than on the total number of parties. This shifts the computational cost balance: the prover still needs to generate commitments for many shares, but the cost of the *MPC emulation* procedure can be reduced significantly.

Related Work MPCitH-based DSAs rely heavily on symmetric primitives and structured linear-algebra computations, which are both attractive operations/targets for hardware acceleration. In recent years, more efficient MPCitH frameworks have been proposed, such as TCitH [FR23, FR25] and VOLEitH [BBdSG+23]. However, research on the acceleration of modern MPCitH-based schemes remains limited, despite being critical for real-world deployment. To the best of our knowledge, hardware designs have only been proposed for (older versions) of `SDitH` [DHSY24] and `Picnic` [KRR+20]. Both designs demonstrate that hardware acceleration of MPCitH-based schemes is feasible, yet resulting in either a high latency or area cost compared to hardware implementations of lattice-based schemes [ZZW+22].

Contributions In this work, we present a compact and efficient hardware implementation of `Mirath`, a Round 2 candidate in the ongoing NIST DSA standardization effort, and its TCitH framework. We propose a combination of algorithmic and hardware-level optimizations, resulting in a unified accelerator for Mirath, supporting the complete `KeyGen`, `SigGen` and `Verify` routines at NIST security level I and V parameter sets. In more detail, our contributions are:

1. An optimized method for performing the Goldreich-Goldwasser-Micali (GGM) tree expansion and subsequent operations suitable for memory-constrained platforms (em-

bedded SW & HW), by generating and immediately consuming the generated leaf nodes *on-the-fly* instead of storing all nodes in memory first. We introduce a novel tree node indexing scheme, enabling efficient traversing through the tree nodes/levels, and a flexible seed buffer construction, reducing the amount of re-computations of common ancestor nodes during GGM tree expansion, which is the main bottleneck.

2. Highly-optimized and parallelized MPC arithmetic units capable of exploiting algorithm-level parallelisms, which achieve high area efficiency by selecting and exploiting HW-friendly `Mirath` parameters. Our novel, high-throughput compute units ensure our design is not bottlenecked by the MPC compute and can process the pseudo-randomly generated data on time.

3. We design and implement high-throughput symmetric primitive engines (SHA3 & AES), as MPCitH-based schemes heavily rely on symmetric primitives. Additionally, we propose an operation scheduling strategy that minimizes signing and verification latency, by ensuring the SHA3 & AES engines and their tightly coupled and highly parallelized arithmetic cores remain maximally active.

4. We evaluate the performance of our compact and efficient design on FPGA and observe significant improvements in terms of area and/or latency compared to state-of-the-art implementations of MPCitH-based schemes most other PQC DSA hardware implementations. Our L1 design occupies 15 400/9 945/42 LUTs/FFs/BRAMs on an Artix-7 FPGA in total and improves total execution time up to a factor 3.7× at 250MHz on FPGA compared to an AVX2 optimized `Mirath` implementation (3 GHz).

The source code of our design is available at https://github.com/KULeuven-COSIC/MIRATH-in-HW.

2 Preliminaries

2.1 Notations

Bold uppercase letters (e.g., $\boldsymbol{S}$) denote matrices and bold lowercase letters (e.g., $\boldsymbol{y}$) denote column vectors, while scalars are written in normal font. Arithmetic is performed over the base field $\mathbb{F}_q$ and its extension field $\mathbb{F}_{q^\mu}$, whose elements are degree-$(\mu-1)$ polynomials over $\mathbb{F}_q$ (i.e., μ coefficients in $\mathbb{F}_q$). For strings, $a\|b$ denotes the concatenation of a followed by b. For matrices, $[\boldsymbol{A}|\boldsymbol{B}]$ denotes horizontal concatenation of $\boldsymbol{A}$ and $\boldsymbol{B}$. $\ll$ and $\gg$ denote left and right shifts, respectively. All matrix generation in `Mirath` is column-major and column-wise byte-aligned: when a column contains a non-multiple-of-8 number of bits, it is padded to the next whole byte. For vectors $\boldsymbol{a}$,$\boldsymbol{b}$ of equal length, $\boldsymbol{a}\cdot\boldsymbol{b}$ denotes their dot product $\sum_i a_i b_i$.

2.2 MPC-in-the-Head and GGM Trees

MPCitH-based signature schemes often rely on *seed trees* to generate the per-party randomness, as it allows to expand a significant amount of pseudo-randomness from a single seed in a structured fashion. A common instantiation is a GGM tree, where a single root seed is expanded into a binary tree by repeated application of a length-doubling PRG based on AES (NIST I) or Rijndael (NIST III & V) [Nat01]. The lowest-level nodes, called *leaf seeds* of the GGM tree (N in total), are used to derive the per-party randomness (polynomials) required by the MPCitH protocol. The GGM tree construction is attractive for reducing the signature size, since it allows the signer to open all-but-one of the tree leaves (hidden index i^*) by revealing a small set of internal[1] nodes, commonly called a *sibling path*.

[1] An internal node is a node in the GGM tree that is neither the root node nor a leaf node.

In Mirath, the N-party MPC computations are repeated τ times in parallel. Instead of relying on τ independent seed trees with N leaves each, Mirath optimizes its signature size by instantiating a single GGM tree with $\tau \cdot N$ leaves, in which the opened paths can be represented by a small amount of internal nodes. This technique is referred to as *batched all-but-one vector commitment* (BAVC), based on [BBM+24]. Mirath further reduces the signature size by imposing an upper threshold T_{open} on the number of opened tree nodes. If a given set of i^* indices would lead to a sibling path longer than T_{open}, the indices i^* are redrawn until an opening whose sibling path is less than or equal to T_{open} is obtained.

2.3 Mirath

Mirath is a quantum attack-resistant MPCitH-based DSA, which relies on the hardness of the MinRank problem [BFG+24]. Several parameter sets are proposed in [AAB+25b], including one with the hardware-friendly[2] modulus $q = 2$. In this work, we utilize the parameter sets as presented in Table 1. τ denotes the number of MPCitH protocol iterations (rounds) performed during the signature algorithm. N denotes the number of parties taking part in each protocol. T_{open} is the maximum number of opened nodes, which is a trade-off between signature size and signing latency.

Table 1: Parameter sets for Mirath-b-fast [AAB+25b].

Instance	MinRank parameters				MPC protocol parameters							Classical bit security
	m	n	k	r	q	μ	ρ	τ	N	T_{open}	w	
Mirath-1b-fast	42	42	1443	4	2	8	16	17	2^8	118	9	158
Mirath-3b-fast	50	50	2024	5	2	8	24	26	2^8	184	10	224
Mirath-5b-fast	56	56	2499	6	2	8	32	36	2^8	244	4	289

We present the top-level key generation, signing, and verification routines in Alg. 1 - 5, respectively. For a complete description of Mirath, we refer the reader to the specification document [AAB+25b].

Key Generation Mirath's public key (pk) and secret key (sk) are compact: they mainly consist of short randomly sampled seeds (Alg. 1). Both can be deterministically expanded into the secret matrices $\boldsymbol{S}$ and $\boldsymbol{C'}$, and public matrix $\boldsymbol{H'}$. In addition, the public vector $\boldsymbol{y}$ can be derived through Algorithm 2. The vectorization operator $\texttt{vec}(\cdot)$ stacks matrix columns in *column-major* order, and $\texttt{Split}(\cdot)$ denotes the fixed partition of a vector into $(\cdot)_A$ and $(\cdot)_B$ parts.

Algorithm 1 KeyGen() [AAB+25b, Alg. 19]

1: $\mathsf{seed_{sk}} \leftarrow \texttt{TRG.Seed}(\lambda)$
2: $\mathsf{seed_{pk}} \leftarrow \texttt{TRG.Seed}(\lambda)$
3: $(\boldsymbol{S},\boldsymbol{C'}) \leftarrow \texttt{ExpandSeedSecretMatrices}(\mathsf{seed_{sk}})$ ▷ $\boldsymbol{S} \in \mathbb{F}_q^{m\times r}, \boldsymbol{C'} \in \mathbb{F}_q^{r\times(n-r)}$
4: $\boldsymbol{H'} \leftarrow \texttt{ExpandSeedPublicMatrix}(\mathsf{seed_{pk}})$ ▷ $\boldsymbol{H'} \in \mathbb{F}_q^{(m\cdot n-k)\times k}$
5: $\boldsymbol{y} \leftarrow \texttt{ComputeY}(\boldsymbol{S},\boldsymbol{C'},\boldsymbol{H'})$ ▷ $\boldsymbol{y} \in \mathbb{F}_q^{(m\cdot n-k)\times 1}$ and ComputeY uses Alg. 2
6: $\mathsf{pk} \leftarrow (\mathsf{seed_{pk}},\boldsymbol{y})$
7: $\mathsf{sk} \leftarrow (\mathsf{seed_{sk}},\mathsf{seed_{pk}})$
8: **return** (pk,sk)

Signature Generation The signing procedure in Mirath (Alg. 3) consists of four steps and produces a non-interactive proof transcript, which is bound to that message and proves knowledge of a secret matrix $\boldsymbol{E}$ such that $\boldsymbol{H}\cdot\texttt{vec}(\boldsymbol{E}) = \boldsymbol{H}\cdot\texttt{vec}(\boldsymbol{SC}) = \boldsymbol{y}$, where $\text{rank}(\boldsymbol{E}) \leq r$.

[2] Over $\mathbb{F}_2$, additions, subtractions, and multiplications reduce to binary operations, enabling high throughput and high clock frequencies with minimal circuit area and few pipeline stages in the arithmetic units.

Algorithm 2 `ComputeY`($\mathbf{S}$,$\mathbf{C'}$,$\mathbf{H'}$) [AAB+25b, Alg. 18]

$\boldsymbol{E} \leftarrow (\boldsymbol{S}|\boldsymbol{SC'})$ ▷ $\mathbf{E} \in \mathbb{F}_q^{m \times n}$
$\boldsymbol{e} \leftarrow \texttt{vec}(\boldsymbol{E})$ ▷ $\boldsymbol{e} \in \mathbb{F}_q^{mn}$ with the entries of $\boldsymbol{E}$ in column-major order
$(\boldsymbol{e}_A, \boldsymbol{e}_B) \leftarrow \texttt{Split}(\boldsymbol{e})$ ▷ $\boldsymbol{e}_A \in \mathbb{F}_q^{(mn-k)\times 1}, \boldsymbol{e}_B \in \mathbb{F}_q^{k \times 1}$
$\boldsymbol{y} \leftarrow \boldsymbol{e}_A + \boldsymbol{H'}\boldsymbol{e}_B$ ▷ $\boldsymbol{y} \in \mathbb{F}_q^{(mn-k)\times 1}$
return $\boldsymbol{y}$

Algorithm 3 `SigGen`(sk,msg) [AAB+25b, Alg. 22]

// Step 0: Initialization.
$(\boldsymbol{H'}, \text{pk}, \boldsymbol{S}, \boldsymbol{C'}) \leftarrow \texttt{DecompressSK}(\text{sk})$ ▷ [AAB+25b, Alg. 21]
salt $\leftarrow \texttt{TRG.Seed}(2\lambda)$
rseed $\leftarrow \texttt{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.
$(\text{base}, \boldsymbol{v}, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \texttt{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, \boldsymbol{S}, \boldsymbol{C'})$ ▷ [AAB+25b, Alg. 11]

// Step 2: Compute the polynomial proof $P_\alpha(X)$.
$\boldsymbol{\Gamma} \leftarrow \texttt{ExpandChallengeMatrix}(h_{\text{sh}})$ ▷ $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$
for e **from** 0 **to** $(\tau-1)$ **do**
  $(\boldsymbol{\alpha}_{\text{mid}}[e], \boldsymbol{\alpha}_{\text{base}}[e]) \leftarrow \texttt{ComputePolynomialProof}(\text{base}[e], \boldsymbol{v}[e], \boldsymbol{S}, \boldsymbol{C'}, \boldsymbol{\Gamma}, \boldsymbol{H'})$ ▷ Alg. 8
$h_{\text{piop}} \leftarrow \texttt{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, \boldsymbol{\alpha}_{\text{mid}}[0], \boldsymbol{\alpha}_{\text{base}}[0], \ldots, \boldsymbol{\alpha}_{\text{mid}}[\tau-1], \boldsymbol{\alpha}_{\text{base}}[\tau-1])$

// Step 3: Open random evaluations of the polynomials $P_S(X)$,$P_C(X)$ and $P_v(X)$.
$(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \texttt{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$ ▷ Alg. 4
$\sigma \leftarrow \texttt{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \boldsymbol{\alpha}_{\text{mid}})$ ▷ [AAB+25b, Alg. 16]
return σ

In Step 0, apart from the sampling of two random values, a salt and the GGM tree root node rseed (see Section 2.2), the secret key sk is expanded into its matrices $\boldsymbol{H'}, \boldsymbol{S}, \boldsymbol{C'}$. In Step 1, the signer commits (τ times) to the witness polynomials $P_S(X) = \boldsymbol{S} \cdot X + \boldsymbol{S}_{\text{base}}$, $P_{C'}(X) = \boldsymbol{C'} \cdot X + \boldsymbol{C'}_{\text{base}}$, and the masking polynomial $P_v(X) = \boldsymbol{v} \cdot X + \boldsymbol{v}_{\text{base}}$ (`BAVC.Commit`, Alg. 6 - 7). The generated commitment hash h_{sh} is used to derive the verifier challenge $\boldsymbol{\Gamma}$, which is used to compute the polynomial proof $P_\alpha(X)$ (Step 2), and the auxiliary value aux (Step 3). Finally, Step 1 also returns a packed value key that contains the nodes of the GGM tree, plus the *commitment* of every *leaf node*.

Algorithm 4 `OpenRandomEvaluations`(key, h_{piop}) [AAB+25b, Alg. 12]

$\texttt{ctr} \leftarrow 0$
`retry:`
$(\{i^*[e]\}_{e<\tau}, v_{\text{grinding}}) \leftarrow \text{ExpandChallengeEvaluationPoints}(h_{\text{piop}}, \texttt{ctr})$ ▷ $i^*[e] \in \{0, \ldots, (N-1)\}$
$\pi_{\text{BAVC}} \leftarrow \text{BAVC.Open}(\text{key}, \{i^*[e]\}_{e<\tau})$
if $\pi_{\text{BAVC}} = \perp$ **or** $v_{\text{grinding}} \neq 0$ **then**
  $\texttt{ctr} \leftarrow \texttt{ctr} + 1$
  goto `retry`
return $(\texttt{ctr}, \pi_{\text{BAVC}})$

In Step 2, the signer computes h_{piop} (hash), which binds the message to the proof values and serves as the input for sampling the openings in Step 3. As described in Alg. 4, the signer samples the hidden indices $i^*[e]$, $e < \tau$ and the grinding vector v_{grinding}, and attempts to open the batched all-but-one commitment using `BAVC.Open` [AAB+25b, Alg. 9]. w is the *grinding* parameter: each opening attempt samples a w-bit value v_{grinding}, and the opening is accepted only if $v_{\text{grinding}} = \mathbf{0}$. Under the standard assumption that v_{grinding} is uniformly distributed, the expected number of attempts is 2^w. If $v_{\text{grinding}} \neq 0$ or if the opening exceeds the allowed size T_{open}, the signer increments ctr and retries with new $i^*[e]$ until a valid

opening π_{BAVC} is obtained.

From this high-level description, it is clear that a hardware implementation of `Mirath` is challenging as it requires a significant amount of runtime flexibility.

Signature Verification Intuitively, the verification routine consists of four steps in which it is verified that the signature σ is a valid MPCitH transcript bound to a message msg under pk. It also checks if (i) the signature contains a valid opening of the batched commitment that fixes the MPC shares (Step 0-1), and that (ii) the recomputed hash value h'_{piop} matches the hash value h_{piop} included in the signature (Step 2-3).

Algorithm 5 `Verify`(pk,σ,msg) [AAB+25b, Alg. 23]

// Step 0: Initialization (parsing and expansion).
1: $(\mathsf{salt},\mathsf{ctr},h_{\mathsf{piop}},\pi_{\mathsf{BAVC}},\mathsf{aux},\boldsymbol{\alpha}_{\mathsf{mid}}) \leftarrow$ `ParseSignature`(σ) ▷ [AAB+25b, Alg. 17]
2: $(\mathbf{H}',\mathbf{y}) \leftarrow$ `DecompressPK`(pk) ▷ [AAB+25b, Alg. 20]

// Step 1: Computing Opened Evaluations.
3: $(\mathsf{evals},\mathsf{point},h_{\mathsf{sh}},v_{\mathsf{grinding}}) \leftarrow$ `ComputeEvaluations`($\mathsf{salt},\mathsf{ctr},h_{\mathsf{piop}},\pi_{\mathsf{BAVC}},\mathsf{aux}$) ▷ [AAB+25b, Alg. 13]

// Step 2: Re-computation of the polynomial proof $P_\alpha(X)$.
4: $\boldsymbol{\Gamma} \leftarrow$ `ExpandChallengeMatrix`(h_{sh}) ▷ $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$
5: **for** e **from** 0 **to** $(\tau-1)$ **do**
6: $\quad \boldsymbol{\alpha}_{\text{base}}[e] \leftarrow$ `RecomputePolynomialProof`($\mathsf{point}[e],\mathsf{evals}[e],\boldsymbol{\Gamma},(\mathbf{H}',\mathbf{y}),\alpha_{\mathsf{mid}}[e]$) ▷ [AAB+25b, Alg. 15]

// Step 3: Verification.
7: $h'_{\mathsf{piop}} \leftarrow$ `Hash`$_2$($\mathsf{pk},\mathsf{salt},\mathsf{msg},h_{\mathsf{sh}},\boldsymbol{\alpha}_{\mathsf{mid}}[0],\boldsymbol{\alpha}_{\text{base}}[0],\ldots,\boldsymbol{\alpha}_{\mathsf{mid}}[\tau-1],\boldsymbol{\alpha}_{\text{base}}[\tau-1]$)
8: **return** $\left(h_{\mathsf{piop}} \stackrel{?}{=} h'_{\mathsf{piop}}\right)$ **and** $\left(v_{\mathsf{grinding}} \stackrel{?}{=} 0\right)$

In Step 1, the verifier re-computes the opened evaluations (`ComputeEvaluations`) to recover (i) the evaluation points $\mathsf{point}[e]$ and corresponding opened evaluations $\mathsf{evals}[e]$ for all repetitions, (ii) the commitment hash h_{sh}, and (iii) the grinding vector v_{grinding}. Internally, this step uses π_{BAVC} to reconstruct all GGM-derived leaf seeds except the hidden ones, regenerates the required pseudorandom shares, and recomputes all commitments of the revealed parties.

3 Proposed Optimization Strategies

In this section, we propose several algorithmic optimizations to improve the performance of Mirath in hardware. We especially focus our optimizations on the most critical parts of Mirath: i) GGM tree expansion, ii) MPC arithmetic, and iii) opening random evaluations.

3.1 On-the-fly GGM Tree Expansion

During signing and verification, the polynomials $P_S(X)$, $P_{C'}(X)$, and $P_v(X)$ are derived from the leaf nodes in the (re-)constructed GGM tree, which is constructed by expanding the internal seed nodes. The signer commits to those polynomials, and the verifier checks that the randomly revealed evaluations are consistent with the provided commitments. Generating and storing the entire GGM tree in memory, consisting of $\tau \cdot N$ leaf seeds and $2 \cdot \tau \cdot N - 1$ nodes in total, before starting processing on the leaf shares leads to high memory overhead and a significant (initial) latency penalty. To illustrate, even at NIST I parameters, the leaf seeds occupy 69.6 KB, and the entire GGM tree occupies 139.2 KB in memory. Hence, we propose to generate $\frac{N}{4}$ leaves at a time and simultaneously consume them *on-the-fly*, as described in Algorithm 7. In this way, the seed data is processed in smaller portions, rather than expanding the full tree ($\tau \cdot N$ leaves) at once. As a result, the required on-chip memory is minimized and, by ensuring the *on-the-fly* seed generation is sufficiently fast, latency is unaffected. More specifically, the proposed optimization improves:

- *Latency*: by generating only $\frac{N}{4}$ leaves at once, which allows us to start processing the leaves in subsequent operations sooner. Routines that consume the leaf seeds can begin as soon as the first leaf group is available, rather than waiting for a full GGM tree expansion. We propose a *hardware-friendly loop scheduling* to ensure that the rate of generating new seeds matches with the consumption rate in the arithmetic modules; therefore, minimizing idle cycles.

- *Memory*: as seed nodes are derived from their parent nodes, which are derived from the root seed. We propose to store a limited (yet configurable) amount of parent nodes in on-chip memory, rather than repeatedly deriving each leaf node from the root seed. To this end, we explore the memory/time trade-off through possible *seed memory configurations* (permanent and buffer storage) and demonstrate how different parameters lead to different memory footprints and latencies.

3.1.1 Optimized Loop Scheduling

A naïve implementation of the BAVC procedure in Mirath (Section 2.2) during signing is shown in Algorithm 6. First, the full GGM tree is constructed using an AES-based PRG (Line 1-3), after which the commitments of each party (N in total, indices i,j) in each repetition (τ in total, index e) are computed (Line 4-8). Finally, the commitment hash h_{com} is computed by hashing (SHA-3) and aggregating all commitments (Line 9-11).

Algorithm 6 `BAVC.Commit`(salt, rseed) adapted from [AAB+25b, Alg. 8]

1: $\mathsf{tree}[0] \leftarrow \mathsf{rseed}$
2: **for** $i \leftarrow 0$ **to** $(\tau \cdot N - 2)$ **do**
3: $\quad(\mathsf{tree}[2i+1], \mathsf{tree}[2i+2]) \leftarrow \mathtt{ExpandSeed}_4(\mathsf{salt}, \mathsf{tree}[i], i)$
4: **for** $e \leftarrow 0$ **to** $(\tau - 1)$ **do**
5: $\quad$**for** $i \leftarrow 0$ **to** $(N/4 - 1)$ **do**
6: $\quad\quad$**for** $j \leftarrow 0$ **to** 3 **do**
7: $\quad\quad\quad\mathsf{seeds}[e][i][j] \leftarrow \mathsf{tree}[(\tau \cdot N - 1) + ((4i+j) \cdot \tau + e)]$
8: $\quad\quad\quad\mathsf{commit}[e][i][j] \leftarrow \mathtt{Commit}(\mathsf{salt}, \mathsf{seeds}[e][i][j], (\tau \cdot N - 1) + (4i+j) \cdot \tau + e)$
9: **for** $j \leftarrow 0$ **to** 3 **do**
10: $\quad h_{\mathsf{commits}}[j] \leftarrow \mathtt{Hash}(\{\mathsf{commit}[e][i][j]\}_{e,i})$
11: $h_{\mathsf{com}} \leftarrow \mathtt{Hash}(\{h_{\mathsf{commits}}[j]\}_j)$
12: $\mathsf{key} \leftarrow \mathsf{tree} \| \mathsf{commit}$
13: **return** $(\mathsf{seeds}, h_{\mathsf{com}}, \mathsf{key})$

Software (CPU) For software implementations on SIMD-capable[3] CPUs, the *party index loop* can be decomposed into a *sequential* loop over the index $i \in \{0,\ldots,\frac{N}{4}-1\}$ and a *parallel* loop over $j \in \{0,\ldots,3\}$, where *party index* $= 4i+j$ (Alg. 6). This ordering ensures that the compiler can take advantage of SIMD datapaths, parallelizing four independent commit computations (Line 8-10).

This loop scheduling requires the platform to support (efficiently) switching between four parallel hashing contexts/states, which would translate to instantiating four SHA3 engines in hardware. However, to perform all four commit computations in parallel, the (leaf) seeds need to be produced sufficiently fast by the AES engine (PRG). Typically, its throughput is limited, thus negating any performance improvement, despite quadrupling the SHA3 area consumption.

[3]SIMD: Single Instruction, Multiple Data. A type of parallel processing commonly implemented in general purpose processors that allows a processor to simultaneously perform the same instruction on multiple data instances.

Hardware (FPGA/ASIC) Instead, we propose to re-order the loop structure so that no more than one hashing context is active at any given time. More specifically, as shown in red in Alg. 7, a hardware-friendly loop ordering of the `BAVC.Commit` routine is shown: the parallel loop j is the most outer loop, such that only $\frac{N}{4}$ parties are processed at a time in each *sub-round* (inner loop i).

Algorithm 7 HW-Friendly `BAVC.Commit`(salt, rseed)

1: $\mathsf{tree}[1] \leftarrow \mathsf{rseed}$
2: **for** $i \leftarrow 1$ **to** $(\tau \cdot N - 1)$ **do**
3: $\quad(\mathsf{tree}[2i], \mathsf{tree}[2i+1]) \leftarrow \mathtt{ExpandSeed}_4(\mathsf{salt}, \mathsf{tree}[i], i)$
4: **for** $j \leftarrow 0$ **to** 3 **do**
5: $\quad$**for** $e \leftarrow 0$ **to** $(\tau - 1)$ **do**
6: $\quad\quad$**for** $i \leftarrow 0$ **to** $(N/4 - 1)$ **do**
7: $\quad\quad\quad\mathsf{seeds}[e][i][j] \leftarrow \mathsf{tree}[(\tau \cdot N) + ((4i+j) \cdot \tau + e)]$
8: $\quad\quad\quad\mathsf{commit}[e][i][j] \leftarrow \mathtt{Commit}(\mathsf{salt}, \mathsf{seeds}[e][i][j], (\tau \cdot N) + (4i+j) \cdot \tau + e)$
9: **for** $j \leftarrow 0$ **to** 3 **do**
10: $\quad h_{\mathsf{commits}}[j] \leftarrow \mathtt{Hash}(\{\mathsf{commit}[e][i][j]\}_{e,i})$
11: $h_{\mathsf{com}} \leftarrow \mathtt{Hash}(\{h_{\mathsf{commits}}[j]\}_j)$
12: $\mathsf{key} \leftarrow \mathsf{tree} \| \mathsf{commit}$
13: **return** $(\mathsf{seeds}, h_{\mathsf{com}}, \mathsf{key})$

Due to the proposed scheduling, for every *sub-round* a sequential stream of $\frac{N}{4}$ leaf seeds is generated and their corresponding commits are computed. This allows a single SHA3 engine to absorb commits continuously using a single hashing context, and only finalize once every τ sub-rounds to obtain $h_{\mathsf{commits}}[j]$ (Line 10). Repeating this for $j \in \{0,1,2,3\}$ yields the four values $h_{\mathsf{commits}}[0],\ldots,h_{\mathsf{commits}}[3]$, which are then hashed to obtain h_{com}. Our approach ensures that it is sufficient to instantiate a single SHA3 engine in hardware, which can be kept constantly active and for which the input seeds can be generated sufficiently fast (AES).

3.1.2 HW-Friendly GGM Seed Tree Indexing and Traversal

In Mirath's one-tree commitment scheme, the binary GGM tree is implemented as an array and seed leaves are indexed as

$$\text{leaf_index} = e + \text{party_index} \cdot \tau. \tag{1}$$

with $e \in \{0,\ldots,\tau-1\}$. In the reference software implementation, a 0-based indexing scheme is used, in which the root node is denoted by zero. As a result, obtaining the child indices of a node i_{parent} is performed as

$$i_{\text{left_child}} = (i_{\text{parent}} \ll 1) + 1, \;\; i_{\text{right_child}} = (i_{\text{parent}} \ll 1) + 2 \tag{2}$$

Reversing the operation to get the parent node from a child can be performed in 2 ways: (i) Checking the LSB of i_{child} to determine whether it is a left *or* a right child, then reversing Eq. (2), or (ii) directly calculating i_{parent} as $i_{\text{parent}} = ((i_{\text{child}} + 1) \gg 1)) - 1$.

To reduce the complexity of traversing the GGM tree, we propose to use a *1-based* indexing scheme instead (red in Alg. 7), such that:

- A single *right shift* of the index of a tree node returns the index of its parent node.
- Accessing a child of a parent node index requires a *left shift* and appending a 0 or 1, for the left or right child.
- For any 2 nodes i_0 and i_1 that belong to the same level in the GGM tree, we can directly obtain their relative distance by counting the number of identical leading bits of their indices. Then, their closest common ancestor i_{anc} can be directly obtained by obtaining the *bitwise and* $(i_0 \,\&\, i_1)$ of the 2 nodes and a right shift by the number of trailing zeros.

The hardware-friendly 1-based indexing enables efficient traversing up and down the seed tree, when searching for the closest, previously-computed ancestor during the on-the-fly tree expansion, and when calculating the sibling path during routine OpenRandomEvaluations (Alg. 4). Critically, our approach simplifies the traversal of the GGM tree without altering the output of the signature.

3.1.3 Optimized Seed Tree Buffering

We propose to not permanently store the entire GGM seed tree in memory ($2\cdot\tau\cdot N-1$ nodes), but instead store only a small portion of the tree - all except K levels - to ensure a balance of memory requirements and (re-) computation time. As such, obtaining a leaf seed consists of identifying the closest stored ancestor, and further expanding it (at most) K levels until a leaf node is reached. Choosing $K=1$ means that only the leaves are not stored permanently, $K=2$ signifies the leaves and their parents need to be re-computed, etc.

The permanent storage is complemented with $K+1$ small buffers capable of storing $\frac{N}{4}$ seeds in total, minimizing on-the-fly computations. We exploit the fact that seed leaves of consecutive repetitions differ in LSB by ensuring our AES-based PRG can derive both child nodes from their common parent simultaneously, by duplicating their datapath. The total memory size (permanent + buffer) required for storing all-except-K levels of the GGM tree is equal to:

$$\text{mem}_{\text{req}}(N,\tau,K)=\begin{cases}2\cdot N\cdot\tau-1, & \text{if } K=0,\\ \frac{N\cdot\tau}{2^{K-1}}+\frac{N}{4}\cdot(K+1)-1, & \text{if } K\neq 0.\end{cases} \tag{3}$$

Figure 1 shows the memory requirements for different choices of K, for all three Mirath security levels (I, III and V). It is clear that increasing K reduces the required on-chip memory significantly up to a certain point ($K=4$), at the cost of increased circuit complexity and required flexibility (e.g. locating closest available ancestor, scheduling seed expansions, and so on...). Similarly, when the amount of parallel repetitions τ is increased[4], the amount of stored tree seeds increases. We select $K=4$ as a good compromise between design/circuit complexity and memory/latency reduction.

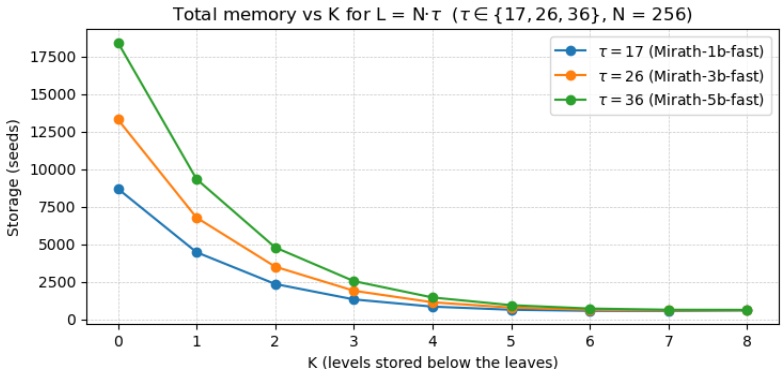


Figure 1: Required GGM node storage as a function of K, evaluated from (3) for the three Mirath parameter sets implemented in this work.

[4]When the security level is kept constant, increasing τ leads to a larger reduction in N, therefore the total size ($2\cdot N\cdot\tau$) of the GGM tree *decreases*. Here, we refer to the case where increasing the security level (without adapting the value of N) leads to an increase in τ and therefore the total size ($2\cdot N\cdot\tau$) of the GGM tree *increases*.

Expanding From Buffered Nodes The combination of the proposed 1-based indexing and seed buffers, leads to efficient expansion of leaf seeds from previously computed ancestor nodes. More specifically, each bit k of the node index indicates if the seed (and its parents/ancestors) are a left or right child node (for 0 or 1, respectively), at level k in the binary GGM tree.

Child nodes are always simultaneously generated by the AES-based PRG and stored in the seed buffers, as the tree is generated from left to right. A child which' parent at level k is a right-hand child (bit at index k is 1), can be derived from that parent at level k. This is because this parent will have been generated together with its left-hand neighbor, from their common parent and stored in the temporary seed buffer. This seed can be selected as closest available ancestor and the seed expansion can be performed downward from there. In the worst case, the ancestor is only available in permanent storage (K levels). Our approach strictly requires bit selecting and right/left rotations, which translate to efficient re-wiring in hardware.

3.2 Accelerating MPC Arithmetic

A second target for optimization in Mirath's signing and verification routines is the repeated computation of the polynomial proof $\boldsymbol{P}_\alpha(X) = \boldsymbol{\alpha}_{\text{mid}} \cdot X + \boldsymbol{\alpha}_{\text{base}}$ for τ MPC repetitions (Step 2 in Alg. 3). In our design, we exploit the fact that this arithmetic is modulo $q=2$ (Table 1) and design highly area-efficient arithmetic modules. We propose a massively parallel compute unit to evaluate all τ MPC iterations of $\boldsymbol{P}_\alpha$ in parallel, heavily reducing latency and not causing a blow-up in area cost, due to the HW-friendly parameters.

During signing specifically, the computation of $\boldsymbol{P}_\alpha(X)$ consists of first computing the pair $(\boldsymbol{\alpha}_{\text{mid}}, \boldsymbol{\alpha}_{\text{base}})$ for all τ repetitions (Algorithm 8). Both computations require 3 steps: i) the construction (and subsequent flattening) of matrices $\boldsymbol{E}_{\text{base}}$ and $\boldsymbol{E}_{\text{mid}}$ (Line 1-3, 7-8), ii) evaluating $\boldsymbol{tmp}$ and $\boldsymbol{tmp'}$ (Line 5 & 10) and iii) multiplying with the challenge matrix $\boldsymbol{\Gamma}$ to obtain both $\boldsymbol{\alpha}$ values (Line 6 & 11).

Algorithm 8 `ComputePolynomialProof`(base, $\boldsymbol{v}$, $\boldsymbol{S}$, $\boldsymbol{C'}$, $\boldsymbol{\Gamma}$, $\boldsymbol{H'}$)

1: $(\boldsymbol{S}_{\text{base}}, \boldsymbol{C'}_{\text{base}}, \boldsymbol{v}_{\text{base}}) \leftarrow \mathsf{base}[e]$
2: $\boldsymbol{E}_{\text{base}} \leftarrow [\boldsymbol{0}_{m \times r} \mid \boldsymbol{S}_{\text{base}} \cdot \boldsymbol{C'}_{\text{base}}]$ ▷ $\boldsymbol{E}_{\text{base}} \in \mathbb{F}_{q^\mu}^{m \times n}$
3: $\boldsymbol{e} \leftarrow \mathsf{vec}(\boldsymbol{E}_{\text{base}})$ ▷ $e \in \mathbb{F}_{q^\mu}^{mn \times 1}$ (column-major)
4: $(\boldsymbol{e}_A, \boldsymbol{e}_B) \leftarrow \mathsf{Split}(\boldsymbol{e})$ ▷ $\boldsymbol{e}_A \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}$, $\boldsymbol{e}_B \in \mathbb{F}_{q^\mu}^{k \times 1}$
5: $\boldsymbol{tmp} \leftarrow \boldsymbol{e}_A + \boldsymbol{H'} \boldsymbol{e}_B$ ▷ $\boldsymbol{tmp} \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}$
6: $\boldsymbol{\alpha}_{\text{base}} \leftarrow \boldsymbol{\Gamma} \cdot \boldsymbol{tmp} + \boldsymbol{v}_{\text{base}}$ ▷ $\boldsymbol{\alpha}_{\text{base}} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$
7: $\boldsymbol{E}_{\text{mid}} \leftarrow [\boldsymbol{S}_{\text{base}} \mid \boldsymbol{S}_{\text{base}} \boldsymbol{C'} + \boldsymbol{S} \boldsymbol{C'}_{\text{base}}]$ ▷ $\boldsymbol{E}_{\text{mid}} \in \mathbb{F}_{q^\mu}^{m \times n}$
8: $\boldsymbol{e'} \leftarrow \mathsf{vec}(\boldsymbol{E}_{\text{mid}})$ ▷ $\boldsymbol{e'} \in \mathbb{F}_{q^\mu}^{mn \times 1}$ (column-major)
9: $(\boldsymbol{e'}_A, \boldsymbol{e'}_B) \leftarrow \mathsf{Split}(\boldsymbol{e'})$ ▷ $\boldsymbol{e'}_A \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}$, $\boldsymbol{e'}_B \in \mathbb{F}_{q^\mu}^{k \times 1}$
10: $\boldsymbol{tmp'} \leftarrow \boldsymbol{e'}_A + \boldsymbol{H'} \boldsymbol{e'}_B$ ▷ $\boldsymbol{tmp'} \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}$
11: $\boldsymbol{\alpha}_{\text{mid}} \leftarrow \boldsymbol{\Gamma} \cdot \boldsymbol{tmp'} + \boldsymbol{v}$ ▷ $\boldsymbol{\alpha}_{\text{mid}} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$
12: **return** $(\boldsymbol{\alpha}_{\text{mid}}, \boldsymbol{\alpha}_{\text{base}})$

In hardware, we propose to parallelize the computations of the pair $(\boldsymbol{\alpha}_{\text{mid}}, \boldsymbol{\alpha}_{\text{base}})$ as their structure and inputs are similar. We also propose specific optimizations for all three steps, especially the matrix-vector product $\boldsymbol{H'} \boldsymbol{e}_B$ in the second step ($\boldsymbol{tmp}$ and $\boldsymbol{tmp'}$), as it accounts for $\pm 73\%$ of the total MPC emulation execution time.

Step 1: Matrix Construction The construction of $\boldsymbol{E}_{\text{mid}}$ requires computing $\boldsymbol{S}_{\text{base}} \boldsymbol{C'} + \boldsymbol{S} \boldsymbol{C'}_{\text{base}}$ and can be computed as $(\boldsymbol{S}_{\text{base}} + \boldsymbol{S})(\boldsymbol{C'}_{\text{base}} + \boldsymbol{C'}) - \boldsymbol{S}_{\text{base}} \boldsymbol{C'}_{\text{base}} - \boldsymbol{S} \boldsymbol{C'}$. This re-ordering is preferred in software, as it re-uses the previously computed $\boldsymbol{S}_{\text{base}} \boldsymbol{C'}_{\text{base}}$ term to minimize latency.

However, we observe that $\boldsymbol{S}, \boldsymbol{C'} \in \mathbb{F}_q$ and $\boldsymbol{S}_{base}, \boldsymbol{C'}_{base} \in \mathbb{F}_{q^\mu}$ and multiplications between $\mathbb{F}_q$ and $\mathbb{F}_{q^\mu}$ are equivalent to multiplying a polynomial by a constant, whereas multipli-

cations of $\mathbb{F}_{q^\mu} \times \mathbb{F}_{q^\mu}$ require multiplying two polynomials. Thus, we directly compute $\boldsymbol{S}_{\text{base}}\boldsymbol{C'} + \boldsymbol{S}\boldsymbol{C'}_{\text{base}}$, alleviating the need for instantiating any $\mathbb{F}_{q^\mu}$-by-$\mathbb{F}_{q^\mu}$ multipliers.

Step 2: Matrix-Vector Multiplication The matrix-vector product $\boldsymbol{H'e}_B$ ($\boldsymbol{H'} \in \mathbb{F}_q$, $\boldsymbol{e'}_B \in \mathbb{F}_{q^\mu}$) is the primary contributor to the MPC emulation cost, where $\boldsymbol{H'}$ is generated in a column-major manner by expanding seed_{pk} (SHAKE). Each output byte contains $\frac{8}{\log_2 q}$ matrix elements, or 8 in the case of $q=2$. We propose to accelerate this operation in hardware through instantiating 8τ optimized $\mathbb{F}_q$-by-$\mathbb{F}_{q^\mu}$ multipliers in parallel, evaluating all τ MPC iterations in parallel and reducing the latency of the $\boldsymbol{tmp}/\boldsymbol{tmp'}$ calculations by the same factor.

Step 3: Challenge Matrix Multiplication Computing $\boldsymbol{\alpha} = \boldsymbol{\Gamma} \cdot \boldsymbol{tmp} + \boldsymbol{v}$ involves multiplications over $\mathbb{F}_{q^\mu}$. Since ρ is small for all parameter sets, this step produces only a small output vector and does not justify replicating expensive $\mathbb{F}_{q^\mu}$-by-$\mathbb{F}_{q^\mu}$ multipliers.

We therefore compute $\boldsymbol{\Gamma} \cdot \boldsymbol{tmp}$ in an element-by-element fashion: for each repetition e and each output index in $\{0,\ldots,\rho-1\}$, we accumulate the corresponding dot product by streaming the matching elements of $\boldsymbol{\Gamma}$ and $\boldsymbol{tmp}$ one pair at a time. The result is then added to the currently accumulated value to form $\boldsymbol{\alpha}$. This schedule keeps the datapath compact while maintaining a high overall throughput for the polynomial evaluation.

3.3 Opening Random Evaluations

The fourth and final step during signing (Alg. 3) consists of opening a *random* set of evaluations π_{BAVC} for the leaves in the tree which will be re-computed by the verifier, thus excluding the τ hidden ones [AAB+25b, Alg. 12]. The set of internal tree nodes (or *sibling path*), whose seeds allow the verifier to recompute all revealed leaves, are included in the signature.

The *sibling path* should contain the least amount of nodes possible, to minimize signature size and bandwidth cost. It is typically constructed by managing a sorted list of node indices, updated through dynamic insert/remove operations while moving through the GGM tree and its hidden leaves. Intuitively, if a node is hidden, one needs to reveal its sibling to the verifier, as revealing the parent node would also reveal the hidden node. If both child nodes can be revealed, their parent node can instead be revealed. We propose a more hardware-friendly construction, as implementing dynamic lists in hardware requires a lot of low-level flexibility.

More specifically, we construct the *sibling path* through a valid-bit table, with each bit corresponding to a node in the GGM tree. For each hidden (leaf) node, each sibling bit is set `HIGH` and the process is repeated for its parent. After processing all τ hidden leaves, the indices with valid-bits `HIGH` are exactly the sibling-path nodes that must be revealed.

Finally, the `Mirath` specification requires the signature to contain the sibling path nodes in increasing node-index order. Our approach does not require any sorting logic but instead can be achieved by performing a single, in-order scan of the valid-bit table starting from the root and appending the signature if bits are '1'. A drawback of the valid-bit approach is that the valid-bit table must be cleared at the start and between retries. However, the clearing is performed during the generation of the next challenge and thus doesn't affect overall performance.

4 Hardware Design

In this section, we describe in detail the architecture and sub-modules of our unified HW implementation of Mirath.

4.1 High-Level Architecture

The main processing elements of our `Mirath` accelerator are a SHA3 engine, an AES-based PRG engine, and an array of fully customized arithmetic cores that implement `Mirath`'s

low-level MPC emulation routines (Figure 2). The AES-based PRG engine is used for GGM expansion, share sampling and opening, and commit generation, controlled by the GGM tree controller. In addition, we integrate two memory elements that are connected to all processing elements: (i) a true dual-port *key scratchpad* that holds signature-related values and (ii) a small true dual-port memory (*commit FIFO*, 8/16 words deep for L1/L5 respectively) used for efficient exchange of commits between the AES engine (producer) and the SHA3 engine (consumer). Finally, a third memory element is connected to the PRG engine, where GGM tree nodes are temporarily stored (buffered), minimizing recomputations during GGM tree expansion.

The arithmetic cores consume outputs produced by the AES and SHA3 engines on the fly, and write back results either to the key scratchpad (when the result becomes part of the signature or other retained state) or to local arithmetic memories. The proposed architecture avoids storing large values in shared memories and keeps shared-memory bandwidth low.

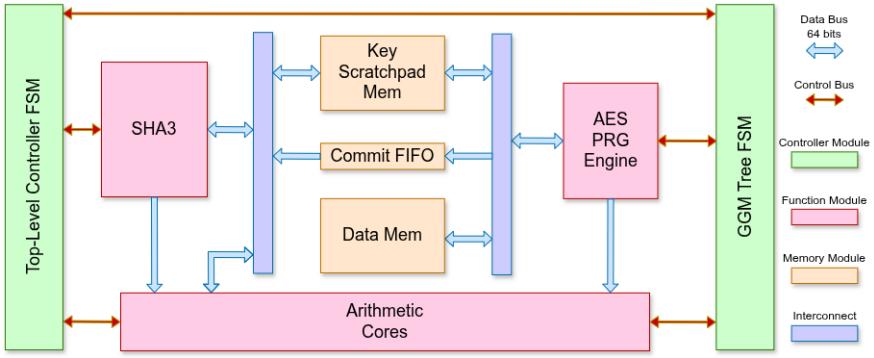


Figure 2: Proposed high-level architecture of `Mirath` in hardware.

4.2 High-Throughput Symmetric Primitive Engines

Symmetric primitives play a crucial role in the construction of MPCitH-based schemes. This section covers our approach for implementing high-throughput symmetric primitive engines, leading to an efficient `Mirath` accelerator.

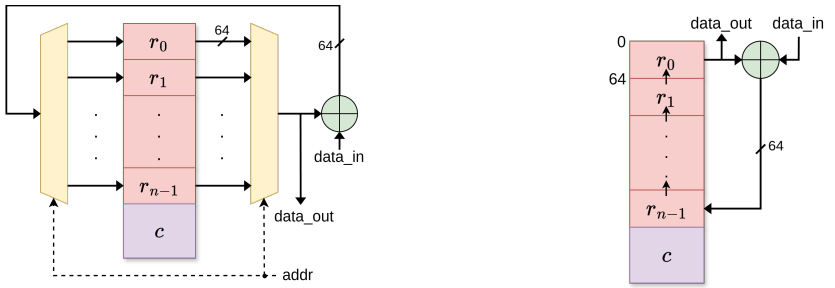
AES/Rijndael `Mirath` heavily relies on AES-ECB (NIST I) or Rijndael (NIST III/V) as a PRG instance for GGM tree expansion, share expansion and commitments[5], making it the main source of latency during signing and verification (Section 2.2 & 3.1). We adapt an AES core from [sec24] and optimize the *encipher* datapath for throughput:

- We implement a *one AES round per cycle* datapath, an improvement of 5× over the reference. The increased S-box area is partially mitigated by removing internal multiplexers and unused control logic, adding only a few hundred LUTs.

- We duplicate the encipher datapath so that two ciphertexts are produced in parallel, enabling the *simultaneous expansion of both child nodes from one parent seed*. Throughput is doubled at the cost of a relatively small area increase, since the key expansion, input registers, and control logic are shared. We recall this is possible due to both child nodes being AES encryptions under the same key and only differing in least significant bit (LSB) in the plaintext.

[5] A Mirath variant that uses SHA-3 for commit generation exists, but is not implemented in this work.

SHA3 `Mirath` uses SHA3 for hashing and as XOF (seed expansion) and thus we design a unified SHA3 engine which supports both, at the security level required by the `Mirath` specification. Our implementation can absorb and squeeze a 64-bit data word per cycle and a full Keccak-$f[1600]$ permutation requires 24 cycles (64 parallel slices) and we store the 1600-bit (25×64-bit) state in a shift register array (Figure 3). Instead of splitting and storing the state at fixed locations in a banked memory architecture, our approach enables flexible access patterns during absorbing and squeezing. More specifically, instead of calculating and updating the target read/write address, the data (Keccak state) is shifted by the correct amount such that the target location/address remains the same. As a result, we avoid explicit address decoding and large selection logic for data access, reducing area consumption and increasing performance.



(a) Memory Bank (b) Shift Register Array

Figure 3: Possible implementations of Keccak-$f[1600]$ (SHA3) ($64b\times24$) state in hardware.

4.3 Compact and Parallelized Arithmetic Units

We develop a set of highly dedicated compute units for accelerating the required arithmetic operations in `Mirath`, shown in Figure 4, which are tightly integrated with the SHA3 and/or AES engines. Instead of instantiating a series of general purpose arithmetic units, we propose a set of area-efficient and high-throughput compute cores that are optimized for `Mirath` parameters and are capable of consuming the data generated by the high-throughput symmetric primitives engines (Section 4.2), in combination with local memory buffers.

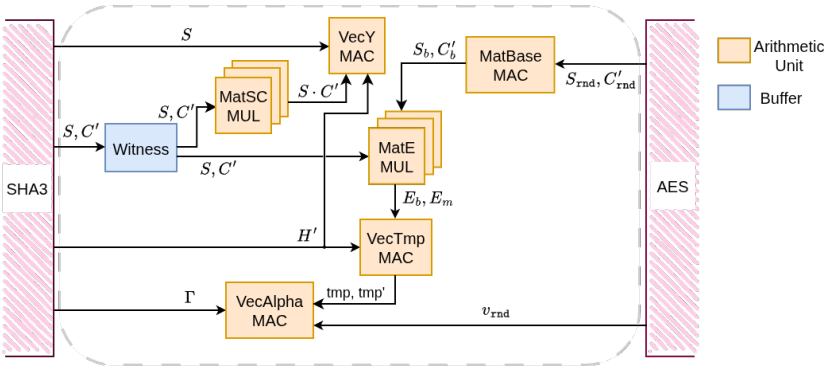


Figure 4: Organization of `Mirath` arithmetic cores, coupled with SHA3 and/or AES engines, and local buffers.

We design our arithmetic cores so that large matrices that are pseudorandomly derived from seed expansions, such as $\boldsymbol{H'}$ and $\boldsymbol{\Gamma}$, can be generated and processed *on-the-fly*, allowing

us to significantly minimize memory requirements, while avoiding any stalling that would occur from memory I/O bandwidth limitations. More specifically, our design includes:

- *Witness Matrix Buffer*: the small matrices ($\boldsymbol{S}$,$\boldsymbol{C'}$) are stored after expansion from $\mathsf{seed_{sk}}$ using the SHA3 engine in `KeyGen` (Alg. 1, Line 3) and `SigGen` (Alg. 3, Line 1) during `DecompressSK`, to be used in `ComputeY` (Alg. 2) and `ComputePolynomialProof` (Alg. 8).
- *MatSC MUL Array ($\mathbb{F}_q \times \mathbb{F}_q$)*: computes $\boldsymbol{SC'}$ on the fly, as required during `ComputeY` (Alg. 2).
- *VecY MAC Unit*: computing vector $\boldsymbol{y} = \boldsymbol{e}_A + \boldsymbol{H'e}_B$ on-the-fly, as required in `KeyGen` (Alg. 1, Line 5) and `SigGen` (Alg. 3, Line 1) during `DecompressSK`.

Additionally, we instantiate the following modules to (re-) compute the polynomials $\boldsymbol{P}_\alpha$, as described in `ComputePolynomialProof` (Alg. 8) (`SigGen`) and `RecomputePolynomialProof` (`Verify`, Alg. 5, Line 6).

- *MatBase MAC Unit*: computing (and storing) $\boldsymbol{S}_{\text{base}}$, $\boldsymbol{C'}_{\text{base}}$, $\boldsymbol{v}_{\text{base}}$ and $\boldsymbol{v}$ during `SigGen` (Alg. 3, Line 4) and `Verify` (Alg. 5, Line 3).
- *MatE MUL Array ($\mathbb{F}_{q^\mu} \times \mathbb{F}_{q^\mu}$)*: computes $\boldsymbol{E}_{\text{base}}$ and $\boldsymbol{E}_{\text{mid}}$ using the buffered witness matrices ($\boldsymbol{S}$,$\boldsymbol{C'}$) and base matrices ($\boldsymbol{S}_{\text{base}}$, $\boldsymbol{C'}_{\text{base}}$), in
- *VecTmp MAC Unit*: the vectors ($\boldsymbol{tmp}$, $\boldsymbol{tmp'}$) are computed by multiplying $\boldsymbol{E}_{\text{base}}$ and $\boldsymbol{E}_{\text{mid}}$ with the matrix $\boldsymbol{H'}$, generated by the SHA3 engine.
- *VecAlpha MAC Unit*: finally, the polynomial components ($\boldsymbol{\alpha}_{\text{base}}$,$\boldsymbol{\alpha}_{\text{mid}}$) are computed by multiplying and accumulating the ($\boldsymbol{tmp}$, $\boldsymbol{tmp'}$) vectors with the challenge matrix $\boldsymbol{\Gamma}$, which is generated on-the-fly.

Each unit contains its own control logic, responsible for memory address generation and accumulation scheduling. The top-level control is only responsible for activating the modules (*start* command), while all fine-grained sequencing is handled within the arithmetic cores.

MatSC MUL Array We propose to compute the matrix product $\boldsymbol{SC'}$ on-the-fly and directly use its entries to construct $\boldsymbol{E} = [\boldsymbol{S} \mid \boldsymbol{SC'}]$, without storing $\boldsymbol{E}$ explicitly in memory. We propose to instantiate a compact array of eight ($\mathbb{F}_q \times \mathbb{F}_q$) multipliers that compute eight elements of $\boldsymbol{SC'}$ every r cycles, with r a matrix dimension and `Mirath` parameter (Table 1). As seen in Figure 5, we schedule the matrix-matrix product such that each multiplier unit computes the product of one column element of $\boldsymbol{S}$ and they all share the element of $\boldsymbol{C'}$. The resulting matrix is large ($m \times n$), but is directly consumed during the computation of $\boldsymbol{y}$ and not stored. In addition, our proposed scheduling is beneficial as the elements of $\boldsymbol{S}$ are generated in a column-wise fashion, enabling direct accessing of 8 consecutive elements. Finally, the achieved throughput is adequate for keeping the cores performing the $\boldsymbol{y}$ evaluation busy while keeping the area footprint low.

VecY MAC Unit We propose to compute $\boldsymbol{y} = \boldsymbol{e}_A + \boldsymbol{H'e}_B$ on the fly. As explained before, $\boldsymbol{H'}$ is large and obtained in a streaming fashion from the SHA3 engine, whereas $\boldsymbol{E}$ is computed on-the-fly by the *MatE MUL Array* and vectorized and split into $\boldsymbol{e}_A$ and $\boldsymbol{e}_B$.

As the size of the accumulated vector $\boldsymbol{y}$ is a non-multiple of the 64-bit chunks of $\boldsymbol{H'}$, using bank-based memory architecture would lead to non-regular access patterns and complex address decoding. For example, the 64-bit SHAKE output word falls within a single byte-aligned column of $\boldsymbol{H'}$, so it can be accumulated in place (Figure 6). However, if the chunk of $\boldsymbol{H'}$ crosses the $\boldsymbol{y}$ column boundary, two disjoint regions of $\boldsymbol{y}$ need to be accessed in the same cycle (Figure 6b).

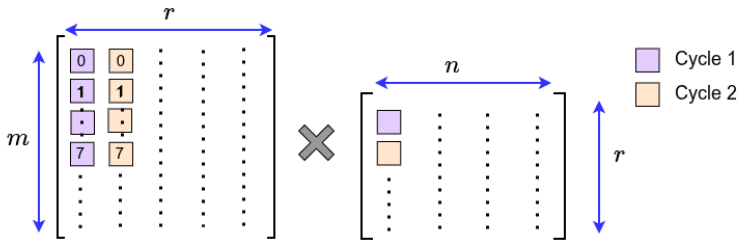


Figure 5: Proposed on-the-fly computation of $\boldsymbol{SC'}$.



(a) No column overlap.

(b) Result contains value belonging to two distinct regions in column accumulator.

Figure 6: Access pattern during accumulation of matrix-vector product (accumulate) $\boldsymbol{y} = \boldsymbol{e}_A + \boldsymbol{H'e}_B$.

Instead, we propose to implement the $\boldsymbol{y}$ accumulator as a *shift register*. Instead of adjusting the read/write location on every cycle, we keep it fixed to the first 64 bits of the register and constantly rotate the data instead. On each Keccak output cycle, the entire $\boldsymbol{y}$ register is rotated by 64 bits, so that the next 64-bit portion to be processed is placed in the fixed access location. The module then adds this word to the input $\boldsymbol{H'e}_B$ term and inserts the updated word back into the end of the $\boldsymbol{y}$ register.

MatBase MAC Unit This unit is responsible for computing and accumulating the base matrices $\boldsymbol{S}_{\text{base}}, \boldsymbol{C'}_{\text{base}}$. They are stored as τ independent memory pairs, one $(\boldsymbol{S}_{\text{base}}[e], \boldsymbol{C'}_{\text{base}}[e])$ per protocol repetition e and used during polynomial evaluations.

The base matrices are computed by multiplying and accumulating the pseudorandom shares $(\boldsymbol{S}_{\text{rnd}}, \boldsymbol{C'}_{\text{rnd}}) \in \mathbb{F}_q$, derived from GGM leaf seeds in the AES engine, with the scalars $\phi(\cdot) \in \mathbb{F}_{q^\mu}$, with $\phi(\cdot)$ a public function [AAB+25b, p. 13]. Since the shares are over $\mathbb{F}_q$, this step consists of computing $\mathbb{F}_q$-by-$\mathbb{F}_{q^\mu}$ products followed by additions in $\mathbb{F}_{q^\mu}$.

MatE MUL Array The buffered matrices $\boldsymbol{S}_{\text{base}}[e]$ and $\boldsymbol{C'}_{\text{base}}[e]$ are streamed $\mathbb{F}_{q^\mu}$ multiplier array, and combined with $\boldsymbol{S}$ and $\boldsymbol{C'}$, streamed from their local buffers to compute $\boldsymbol{E}_{\text{mid}} = [\boldsymbol{S}_{\text{base}} \,|\, \boldsymbol{S}_{\text{base}}\boldsymbol{C'} + \boldsymbol{S}\boldsymbol{C'}_{\text{base}}]$ and $\boldsymbol{E}_{\text{mid}} = [\boldsymbol{0} | \boldsymbol{S}_{\text{base}}\boldsymbol{C'}_{\text{base}}]$ on-the-fly.

We propose a parallelized design, such that all τ memory pairs can be read in parallel to support the τ-lane parallel polynomial evaluation: the multiplier array computes one element of $\boldsymbol{E}_{\text{base}}$ and one element of $\boldsymbol{E}_{\text{mid}}$ at a time. This throughput is sufficient, since each element of $\boldsymbol{e}_B$ and $\boldsymbol{e'}_B$ is used while processing an entire column of $\boldsymbol{H'}$, in the *VecTmp MAC* unit.

VecTmp MAC Unit The $\boldsymbol{tmp}$ accumulator module consists of $2 \cdot \tau$ parallel BRAM-based accumulators and computes the intermediate vectors $\boldsymbol{tmp}$ and $\boldsymbol{tmp'}$ that dominate the computational cost of the polynomial proof evaluation (Algorithm 8), as discussed in Section 3.2. They are computed without $\boldsymbol{H'}$ or $\boldsymbol{e}_B/\boldsymbol{e'}_B$ being stored, but instead being streamed from the SHA3 engine and local buffers, respectively. The unit also contains eight parallel $\mathbb{F}_q$-by-$\mathbb{F}_{q^\mu}$ multipliers to multiply both operands, before accumulation.

Each accumulator ($2\cdot\tau$ in total) holds a full $(mn-k)\times 1$ vector over $\mathbb{F}_{q^\mu}$, so the storage width is $\mu\cdot\log_2(q)$ bits per element. This MAC unit consumes about 70% of the used BRAMs of our entire implementation.

VecAlpha MAC Unit As seen in Alg. 8 (Line 6 & 11), the $\boldsymbol{\alpha}$ vectors are the sum of ($\boldsymbol{v}$,$\boldsymbol{v}_{\text{base}}$) and the product $\boldsymbol{\Gamma}\cdot\boldsymbol{tmp}$ (τ parallel repetitions). First, the module acts as the accumulator for the masking vectors $\boldsymbol{v}_{\text{base}}[e]$ and $\boldsymbol{v}[e]$: as each share is generated, its contribution is accumulated and stored as initial states $\boldsymbol{\alpha}_{\text{base}}[e]\leftarrow\boldsymbol{v}_{\text{base}}[e]$ and $\boldsymbol{\alpha}_{\text{mid}}[e]\leftarrow\boldsymbol{v}[e]$. Second, the multiplications $\boldsymbol{\Gamma}\cdot\boldsymbol{tmp}$ and $\boldsymbol{\Gamma}\cdot\boldsymbol{tmp'}$ are accumulated. These products involve $\mathbb{F}_{q^\mu}$-by-$\mathbb{F}_{q^\mu}$ multiplications, but account for only a small part of the total MPC emulation cost. As such, we compute them in a sequential manner: for each repetition e, the module updates $\boldsymbol{\alpha}_{\text{base}}[e]$ and $\boldsymbol{\alpha}_{\text{mid}}[e]$ element-by-element, while still processing all τ repetitions in parallel.

5 Operation Scheduling

In this section, we discuss how `Mirath` routines are scheduled on the proposed architecture, such that overall performance is maximized and all sub-modules are kept maximally active.

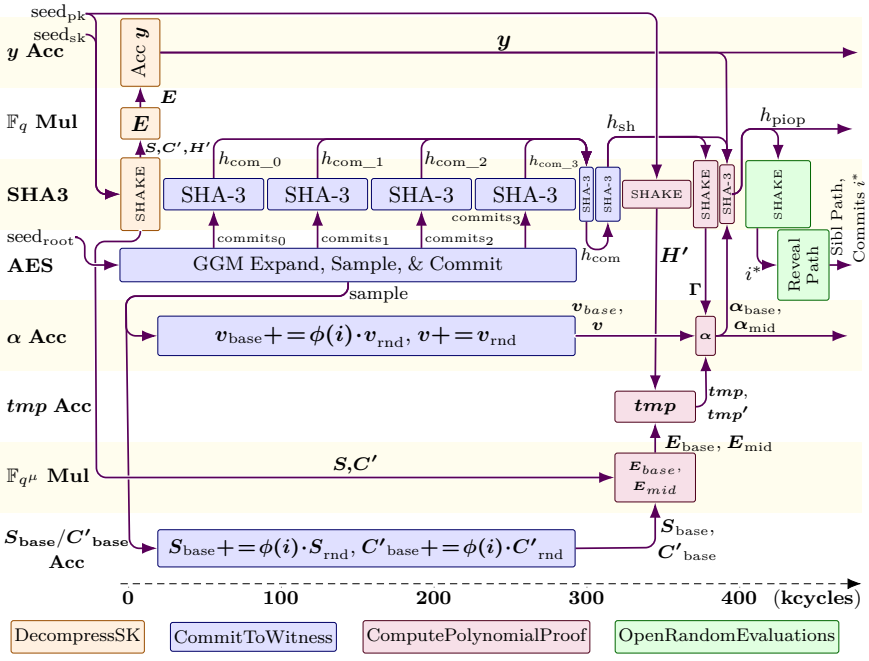


Figure 7: Proposed `Mirath` signature generation scheduling in hardware.

Signature Generation As discussed earlier, `Mirath` signing consists of four stages (Alg. 3), indicated by different colors in Figure 7. We propose to commence the GGM tree expansion (AES) and commit computations (SHA3) as soon as possible, due to them heavily relying on symmetric primitives and accounting for the largest latency. More specifically, Step 1 (`CommitToWitness`) requires a significant amount of AES PRG operations and is scheduled in parallel with the secret key expansion (SHA3) in Step 0 (`DecompressSK`).

After DecompressSK completes, the AES datapath starts filling the commit FIFO buffer, which are are directly processed by the SHA3 engine to produce the hash commits.

Simultaneously, the arithmetic modules start accumulating $\boldsymbol{S}_{\text{base}}$, $\boldsymbol{C'}_{\text{base}}$, $\boldsymbol{v}_{\text{base}}$, and $\boldsymbol{v}$. At the end of Step 1, the SHA3 engine finalizes h_{sh}, which is used to derive the verifier's challenge $\boldsymbol{\Gamma}$.

In Step 2 (Algorithm 8), the polynomial proof is computed: due to the parallel design of our arithmetic cores, this step contributes less than 20% of the total signing latency. After the polynomial coefficients $\boldsymbol{\alpha}_{\text{base}}$ and $\boldsymbol{\alpha}_{\text{mid}}$ are obtained, the signer hashes all proof values to compute h_{piop}.

Finally, Step 3 (OpenRandomEvaluations, Algorithm 4) repeatedly attempts to obtain a valid opening until one is found. Due to its stochastic nature, the latency of Step 3 can vary significantly between signatures. For parameter sets L1 and L3, where the grinding parameter w is large, Step 3 contributes about 35% of the total signing latency on average. Overall, our scheduling ensures the critical modules (SHA3, AES, accumulators) are stalled to a minimal degree, minimizing the overall latency.

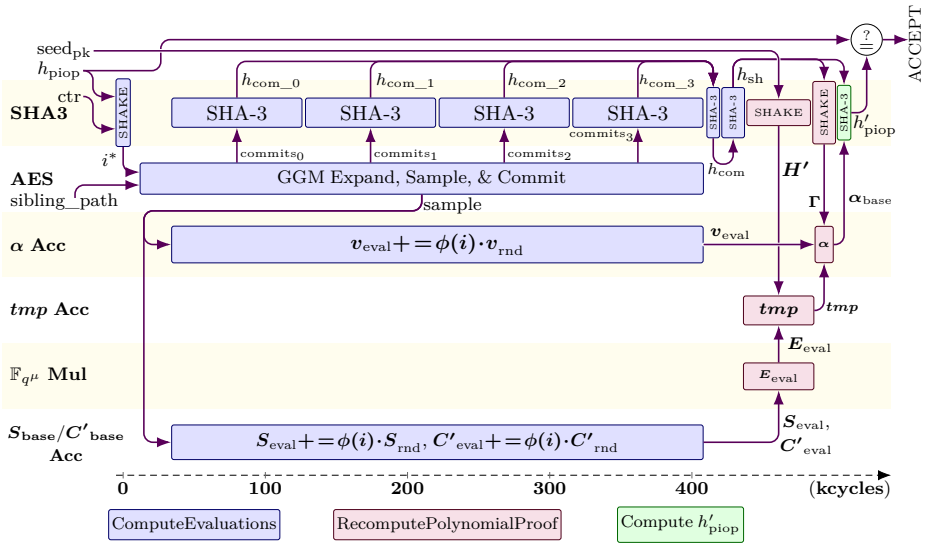


Figure 8: Proposed `Mirath` signature verification scheduling in hardware.

Signature Verification Figure 8 shows the operation scheduling of the verification procedure (Algorithm 5), with the tree main steps explicitly indicated in different colors. Step 0 consists of operations performed on-the-fly, such as expanding seed_{pk} to obtain $\boldsymbol{H'}$, and is therefore absorbed into the rest of the schedule.

In Step 1, the verifier performs partial GGM tree re-expansion to recover the revealed leaf seeds, using the provided *sibling path* and expands the corresponding party shares to build $\boldsymbol{S}_{\text{eval}}$, $\boldsymbol{C'}_{\text{eval}}$, and $\boldsymbol{v}_{\text{eval}}$, and the associated commitments. Compared to Step 1 of signing, this step takes about 35% more time on average because verification combines partial tree reconstruction with the on-the-fly expansion, increasing control complexity, and as a result short stalls occur in the AES datapath. Finally, the SHA3 engine finishes `ComputeEvaluations` by re-computing h_{sh}.

In Step 2, the verifier runs `RecomputePolynomialProof` to recover $\boldsymbol{\alpha}_{\text{base}}$, using the massively parallelized arithmetic units. This step requires the same number of cycles as Step 2 of signing and contributes about 15% of the total verification latency. In Step 3, the verifier computes h'_{piop} (SHA3) and compares it to the h_{piop} value included in the signature. If the values match, the verifier accepts the signature.

6 Implementation and Evaluation

We simulate, synthesize and implement our proposed design (SystemVerilog RTL) using Xilinx Vivado 2024.2, targeting the Xilinx Artix-7 XC7A200T-3 FPGA. In this section, we evaluate the performance of our hardware implementation and compare with related work.

6.1 Resource Utilization

Table 2 presents detailed area utilization metrics at NIST level I and V for the sub-modules in our design, described above. 49.6/68.1% (L1/L5) of the design's LUTs are used in the arithmetic modules, of which 23.2/26.3% are used for multiple distributed-memory instances (depth ≤ 64), used to accumulate small matrices and vectors. Further, the symmetric primitives (SHA3 + AES engine) occupy 37.9/25.5% of total LUTs and 34.5/22.7% of total FFs for L1/L5, respectively.

Table 2: Resource utilization (LUT, FF, DSP, BRAM) of `Mirath` in hardware, including all sub-modules.

Module	**LUT** L1 / L5	**FF** L1 / L5	**DSP** All levels	**BRAM** L1 / L5
Top-level Control (FSM)	958 / 1,007	1,003 / 1,203	0	0
GGM Tree Control (FSM)	929 / 909	371 / 344	0	1 / 2
SHA3 Engine	3,757 / 3,593	1,978 / 1,973	0	1
AES/Rijndael Engine	2,073 / 4,163	1,278 / 2,162	0	0
Arithmetic Cores (Total)	7,639 / 20,699	4,751 / 12,477	0	34 / 73
Data Memory	0	0	0	4 / 16
Key Scratchpad Memory	0	0	0	2 / 4
Commit FIFO	44 / 44	64 / 64	0	0 / 0
Total	15,400 / 30,415	9,445 / 18,223	0	42 / 96

Overall, due to our memory usage optimizations, only 9.5/13.3% (for L1/L5, resp.) of our BRAM usage comes from the data memory for all security levels. In contrast, 81.0/66.7% (for L1/L5, resp.) of the BRAM consumption comes from the $\boldsymbol{tmp}$ accumulator, which stores the intermediate $\boldsymbol{tmp}$ and $\boldsymbol{tmp'}$ vectors, both in $\mathbb{F}_{q^\mu}^{(mn-k)\times 1}$, for all τ protocol repetitions. Although these memories dominate the BRAM allocation, the required number of BRAMs is determined by the memory I/O bandwidth constraints rather than by the amount of stored data.

L1 vs. L5 As expected, we observe a significant difference in resource consumption by the arithmetic modules between the different parameter sets, as the number of parallel protocol instances τ for L5 is more than double that of L1. In addition, the widths of the MatBase core's MAC units are doubled to match the higher througput of the Rijndael-256 engine (compared to the AES-128 engine used at NIST L1). Finally, the AES/Rijdael datapath also shows non-trivial increases (2.01×) in size as we move from the AES-128 (L1) to the Rijndael-256 (L5) datapath.

6.2 Performance

Our FPGA hardware accelerator of `Mirath` enables `KeyGen` to complete in 63.4μs, `SigGen` in 2.181ms (avg.) and `Verify` in 1.969ms at 254 MHz for NIST L1. The signing latency varies across signatures, due to the OpenRandomEvaluations step: for NIST I, the larger grinding parameter w significantly increases the expected number of opening attempts. For NIST V, the grinding parameter w is small; however the choice of T_{open} is more conservative to

decrease the signature size. This leads to a higher probability of a given set of i^* indices being rejected, increasing the average signing latency. At L5 parameters, we observe a slight drop in the maximal operating frequency (218 MHz), which stems from the increased complexity of the Rijndael-256 engine vs. the AES-128 engine[6], as well as the increased complexity of increasing complexity of place-and-route as the size of the circuit increases.

Comparison with AVX2 Optimized Software Compared to the AVX2 optimized C implementation of `Mirath` [AAB+25b], our hardware design reduces the average cycle count for `KeyGen` by 31.1 - 33.0×, for `SigGen` by 17.7 - 51.1×, and 11.6 - 42.7× for `Verify` at NIST L1 and L5, respectively. Compared to the verification routine, signature generation has less runtime variability and offers more room for parallelization, which is why the cycle reduction factor is larger. This is especially pronounced for NIST V, since the inherently sequential grinding step contributes much less to the total runtime. Here, our FPGA implementation improves execution time by a factor 3.7×, and a factor 444.7× compared to `Mirath` software implementations (AVX2 & reference). Although not highly parallelizable, key generation is also significantly accelerated by our design due to the use of our highly optimized SHA3 engine. Overall, these improvement factors can be significantly increased by comparing with the performance of our accelerator on a dedicated ASIC, as FPGA platforms are inherently limited in maximal clock frequency and thus limiting execution time. We consider this exploration out-of-scope and leave it as future work.

Comparison with Related Works We now compare our design with state-of-the-art hardware implementations of post-quantum digital signature algorithms (Table 3). Due to differences in target platforms, optimization goals, and security levels, direct comparisons across designs are not always meaningful. To keep the comparison as fair as possible, we prioritize works targeting Artix-7 devices, since staying within a single FPGA family improves comparability. We include implementations of schemes standardized in NIST's first PQC process, as well as candidates from the "additional signatures" process. We also include PICNIC, as it is one of the few available hardware implementations of MPCitH signatures and provides a useful reference point for recent MPCitH protocol improvements.

A direct latency comparison against current PQC standards is inherently difficult due to fundamental algorithm-level differences (lattice-based vs. MPCitH-based). The design by Norga et al. [NKV25], implementing the lattice-based `ML-DSA`, has a lower latency than `Mirath`. However, our design is more area-efficient: even at L5, our design does not use DSPs and requires only 55.4% of resources compared to [NKV25]. Compared to the L1 `Falcon` implementation by Ouyang et al. [OZZ+25], which only supports `SigGen`, `Mirath` signing is 3.41× slower while our design requires 5.13× fewer resources. The `SPHINCS+` design [ALCZ20] (hash-based) offers similar signing latency, while consuming substantially more FPGA resources.

Further, we compare with code-based signature implementations `CROSS` [KAB+25] and `LESS` [BWMG23]. Our design significantly outperforms LESS in both area, by a factor 2.59 - 2.61× and latency, by a factor 10.1 - 100.6×, for different security levels. Compared to `CROSS`, our implementation is more compact ($4.6-30.4\%$) but has a higher total computation time ($4.57-5.13\times$). We also include the performance of state-of-the-art hardware implementations of multivariate-based DSAs, `MAYO` [HSK+24] and `UOV` [BCD+25]. Our L1

[6]The critical path at L5 is in the Rijndael-256 key-expansion datapath rather than in the encryption datapath itself. Generating the next 256-bit round key requires two dependent SubWord evaluations: one on a word from the previous key state, and another on an intermediate word that has just been computed from the result of the first. To avoid introducing additional latency, we do not add extra pipeline stages. This makes the Rijndael-256 key-expansion datapath noticeably deeper than the AES-128 key-expansion datapath and is the main reason for the lower maximal frequency at L5. Pipelining the Rijndael-256 engine would reduce the frequency gap between our L1 and L5 designs, but would also increase circuit complexity and the latency of the Rijndael-256 datapath, which is already the dominant contributor to the overall latency.

Table 3: Comparison of our `Mirath` hardware design with state-of-the-art PQC DSA HW (& SW) implementations.

Scheme	Parameter Set	Resources LUT/FF/DSP/BRAM	Area[a]	Freq. (MHz)	KeyGen (Mcycles/ms)	Sign (Mcycles/ms)	Verify (Mcycles/ms)	ATP[a] Area×s
Mirath (This work)	1b-fast	15,400/9,445/0/42.0	28,000	254	0.0161/0.0634	0.554/2.181	0.5/1.969	118.0
	5b-fast	30,415/18,223/0/96.0	59,215	218	0.0607/0.2784	1.3726/6.296	1.2116/5.558	718.4
Mirath[b] [AAB+25a, AAB+25b]	1b-fast	–/–/–/–	–	3000	1.8/0.60	1600/533.3	2600/866.67	–
	5b-fast	–/–/–/–	–	3000	6.5/2.17	8400/2800.0	13000/4333.3	–
	1b-fast-AVX2	–/–/–/–	–	3000	0.5/0.17	9.8/3.27	5.5/1.83	–
	5b-fast-AVX2	–/–/–/–	–	3000	2.0/0.67	70.1/23.37	51.7/17.23	–
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	10.07
	L5	54,942/29,309/16/29.0	65,242	230	0.014/0.0609	0.038/0.1652	0.0146/0.0635	18.89
Falcon [OZZ+25]	L1	79,065/46,389/226/45.0	115,165	65	–/–	0.16/0.64	–/–	–
SPHINCS+ [ALCZ20]	128s	48,231/72,514/0/11.5	51,681	250/500	–/–	–/12.40	–/0.07	644.5
	128f	47,991/72,505/1/11.5	51,541	250/500	–/–	–/1.01	–/0.16	60.3
	256s	51,130/74,576/1/22.5	57,980	250/500	–/–	–/19.30	–/0.14	1,127.1
	256f	51,009/74,539/1/22.5	57,859	250/500	–/–	–/2.52	–/0.21	158.0
CROSS [KAB+25]	RSDP-1-f	26,741/11,678/0/45.0	40,241	118	–/0.035	–/0.478	–/0.409	37.1
	RSDP-5-f	34,626/15,331/0/91.5	62,076	113	–/0.028	–/1.20	–/1.14	147.0
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	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	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	3,079.6
	L5-b	104,300/76,700/0/167.5	154,550	143	0.134/0.93	129.89/909.20	129.726/908.08	281,004.4
	L5-s	104,300/76,700/0/167.5	154,550	143	0.247/1.73	87.16/610.13	87.013/609.09	188,697.8
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	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	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	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	2,155.1
	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	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	8,052.4
	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	6,589.7
	V-pkc+skc	77,352/38,217/4/359.0	185,452	92.5	39.1/422.70	3.31/35.784	1.92/20.757	88,876.2
SDitH [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	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	23,651.5
	L5-GF256	23,323/14,962/0/520.5	179,473	164	0.083/0.51	17.48/106.60	24.940/152.10	46,521.2
	L5-GF251	34,456/31,409/472/521.5	238,106	164	0.086/0.53	45.28/276.07	30.110/183.57	109,569.2
PICNIC [KRR+20]	L1	90,337/23,105/0/52.5	106,087	125	–/–	0.03/0.25	0.030/0.24	52.0
	L5	167,530/33,164/0/98.5	197,080	125	–/–	0.15/1.24	0.147/1.17	475.0

[a] Area-Time-Product (ATP), with area measured as $\#LUTs + 100 \times \#DSPs + 300 \times \#BRAMs$ [YCH22].
[b] Intel i9-13900K (3GHz), 64GB RAM, GCC v11.4.0

design consumes 76.6% less area than `MAYO`, but is 8.54× slower. Compared to the fastest `UOV` implementation (Ip-classic), our design is 8.9× faster and reduces the ATP by 21× for NIST L1. Compared with the most efficient `UOV` implementations (better in terms of ATP), our design is 10.9×/39.5× faster and 18.3×/123.7× better at ATP for NIST L1/L5, respectively.

Finally, we compare the performance of our accelerator against the MPCitH-based `SDitH` [DHSY24] implementation. At L1, our design requires slightly fewer LUT & FF resources (3%) while reducing BRAM consumption by a factor 3.92×. At L5, our design requires 30.4% more LUTs and 21.8% more FFs, while BRAM consumption is reduced by 5.42×. Our accelerator achieves a total speed-up in latency of 22.4× / 21.4×, while reducing ATP by 52.7× / 64.8× for L1 / L5 respectively. Compared to `PICNIC` [KRR+20], our design is 3.33× more compact at L5, while total execution time is 5.0× higher. This difference mainly comes from algorithm-level trade-offs: Mirath is optimized for much smaller signatures at the expense of higher latency.

7 Impact and Discussion

Application to Other Schemes The proposed design methodology and optimizations can be directly generalized and extended to other state-of-the-art MPCitH schemes. In particular, `SDitH` (v2.0) has a similar `SigGen` and `Verify` structure as `Mirath` and relies on the same core building blocks: (i) A GGM-style seed tree, expanded via a block-cipher PRG, where each internal node produces two child seeds through two encryptions under the same key and

closely related plaintexts; (ii) a subsequent commitment generation derived from these seeds, followed by Fiat–Shamir hashing to obtain challenge values; (iii) a large $\tau \cdot N$-leaf GGM tree of which all-but-τ leaves are revealed via a sibling path, followed by the same partial GGM tree expansion; (iv) an `OpenRandomEvaluations` step that repeatedly attempts to obtain a valid opening ($v_{\text{grinding}} = 0$ and *sibling path length* $< \text{T}_{\text{open}}$); (v) SDitH-v2's MPC emulation phases are also dominated by structured linear algebra operations over the base field $\mathbb{F}_q$ and its extension field. While the exact operations are scheme-specific, these computations benefit from the same on-the-fly multiplier/accumulator structures proposed in this work.

As a result, both the algorithmic and hardware-level optimizations introduced in this work can be used directly to accelerate `SDitH` and other MPCitH-based schemes. In addition, most of our algorithm-level optimizations can be extended to software implementations, which would not only improve performance, but also make MPCitH-based schemes more practical for resource-constrained embedded devices.

Conclusion This work presents a compact and high-performance hardware implementation of `Mirath`, an MPCitH-based post-quantum digital signature schemes. To the best of our knowledge, it is the first to implement an MPCitH-based scheme that is among the current candidates of NIST's "additional signatures" call, incorporating recent improvements to the MPCitH framework. Our results indicate that modern MPCitH-based schemes can achieve good performance through hardware acceleration (area and latency) and allow for interesting hardware-oriented trade-offs, making them strong candidates for standardization.

As is the case for many hardware implementations of PQC digital signature schemes, overall latency is dominated by symmetric primitives (hashing, XOF, PRG). In the case of MPCitH-based schemes, a significant amount computational complexity related to MPC emulation is introduced. In this work, we: (i) design high-throughput, area-efficient symmetric primitive engines (AES and SHA3) and an architecture which is able to efficiently manage the significant amount of run-time flexibility required while navigating GGM trees (ii) design tightly-coupled, massively parallel arithmetic units, (iii) propose algorithm-level optimizations to exploit hardware-level parallelism and require a minimal amount of on-chip memory, (iv) propose on-the-fly scheduling and computation, that balances the workload between all sub-modules. As a result, we achieve a substantial performance improvement compared to optimized software implementations (3.7 - 444.7×) and outperform state-of-the-art hardware implementations of PQC DSAs in terms of area and/or latency, including MPCitH-based `SDitH`, for which we improve the area-time product up to a factor 23.0×. Overall we demonstrate that, through careful architectural and implementation-level design, MPCitH-based DSAs can be significantly accelerated in hardware, towards real-world requirements.

Acknowledgments

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), Cybersecurity Research Program Flanders (CRPF) with reference number VOEWICS02, and Intel Corporation.

References

[AAB+25a] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath signature

scheme (v2.0). Technical report, February 2025. Available at https://pqc-mirath.org/assets/downloads/mirath_v2.0.pdf.

[AAB+25b] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath signature scheme (v2.1), September 2025. Available at https://pqc-mirath.org/assets/downloads/mirath_v2.1.0.pdf.

[ABB+25a] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, Matthieu Rivain, Jean-Pierre Tillich, and Keita Xagawa. Perk (v2.0), February 2025. Available at https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/perk-spec-round2-web.pdf.

[ABB+25b] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Romaric Neveu, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vinçotte. Ryde signature scheme (v2.0), February 2025. Available at https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/ryde-spec-round2-web.pdf.

[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.

[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 (v2.0), 2025. Available at https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/mqom-spec-round2-web.pdf.

[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.

[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.

[BFG+24] Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, and Matthieu Rivain. Dual support decomposition in the head: Shorter signatures from rank SD and MinRank. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part II*, volume 15485 of *LNCS*, pages 38–69. Springer, Singapore, December 2024.

[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.

[CDG+17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1825–1842. ACM, 2017.

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

[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.

[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.

[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.

[Nat01] National Institute of Standards and Technology. Advanced encryption standard (aes). Federal Information Processing Standards Publication NIST FIPS 197-upd1, National Institute of Standards and Technology, Washington, D.C., November 2001. Updated May 9, 2023.

[Nat16] National Institute of Standards and Technology (NIST). Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. NIST CSRC Technical Report NIST-CSRC, NIST Computer Security Resource Center, December 2016. Final Call for Proposals, Post-Quantum Cryptography.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Federal Information Processing Standards Publication 204, National Institute of Standards and Technology, Washington, DC, August 2024. Published August 13, 2024.

[Nat24b] National Institute of Standards and Technology. Stateless hash-based digital signature standard. Federal Information Processing Standards Publication 205, National Institute of Standards and Technology, Washington, DC, August 2024.

[NIS23] 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. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/additional-signatures-2023.

[NIS24] NIST. PQC digital signature second round announcement. Online. Accessed 10th November, 2024, 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.

[sec24] secworks. AES: on_the_fly_keygen branch. https://github.com/secworks/aes/tree/on_the_fly_keygen, 2024. Accessed: 2024-05-22.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[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.

[ZZW+22] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for crystals-dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, Issue 1:270–295, 2022.