Breaking the Myth of MPCitH Inefficiency: Optimizing MQOM for Embedded Platforms

Ryad Benadjila and Thibauld Feneuil

CryptoExperts, Paris, France
{ryad.benadjila,thibauld.feneuil}@cryptoexperts.com

Abstract. Signature schemes based on the MPC-in-the-Head (MPCitH) paradigm play an important role in enabling cryptosystems founded on a wide diversity of hardness assumptions. While the design of such schemes is currently stabilizing, providing efficient implementations on embedded devices remains a critical challenge, as MPCitH frameworks are known to manipulate large data structures and to rely heavily on symmetric primitives.

In this work, we present a highly optimized implementation of the NIST candidate MQOM (version 2) targeting embedded microcontrollers. Our implementation significantly outperforms existing MPCitH implementations on such platforms, both in terms of memory footprint and execution time. In particular, for the L1 parameter set, we can achieve an SRAM usage below 10 KB, including the key and signature buffers, while preserving practical signing and verification performance (on the order of a few hundred megacycles). We further explore time-memory trade-offs, achieving execution times below 100 Mc for certain variants at the cost of an additional 5-10 KB of memory.

We also provide the first memory-friendly implementation of the one-tree technique, which is used to reduce signature sizes in several MPCitH-based schemes. This enables a comparative analysis of the implementation costs of correlated trees (used in MQOM) versus the one-tree technique (used in other candidates). We then demonstrate how streaming and precomputation techniques can further mitigate the impact of the running time and the signature size. For instance, these approaches enable overlapping computation with data reception, for example by starting computations before all inputs are available, thereby reducing overall latency.

Keywords: Post-quantum signatures · MPC-in-the-head · MQOM · Embedded

1 Introduction

Introduced in 2007 by [IKOS07], the MPC-in-the-Head (MPCitH) paradigm is a versatile framework for constructing efficient digital signature schemes from generic one-way functions or hard problems. Over the past few years, MPCitH frameworks have evolved significantly, leading to increasingly compact signatures. The most recent frameworks [BBD+23b, FR25] enable the construction of signatures of size roughly 2.5-5 KB, whose security relies on the hardness of very conservative problems, such as recovering AES keys [BBB+24c], solving unstructured multivariate quadratic systems [BBFR24], or solving unstructured syndrome decoding instances [ABB+24].

This flexibility in supporting a wide range of security assumptions is particularly attractive in the current context, as the cryptographic community seeks to diversify the hardness assumptions underlying quantum-resilient cryptography. Nine signature schemes based on the MPCitH paradigm have been submitted to the NIST process for standardizing additional post-quantum signature schemes. This process is currently in its

second round, in which six out of the fourteen remaining candidates rely on the MPCitH paradigm [ABC+24].

As the design of MPCitH schemes is now stabilizing, the next important step is the development of optimized implementations. In particular, understanding their concrete performance on embedded devices remains a critical challenge. Existing implementations of current NIST MPCitH-based candidates typically target desktop-class architectures or rely on memory-intensive strategies that do not scale well with constrained microcontrollers.

In this work, we focus on the NIST candidate MQOM (for "MQ on my Mind") [BBFR24], a Threshold-Computation-in-the-Head (TCitH)-based signature scheme whose security relies on the hardness of solving unstructured multivariate quadratic systems. We challenge the widespread assumption [KPR+, BBB+24d, STW25] that MPCitH-based schemes are inherently inefficient on constrained platforms. By presenting a highly optimized implementation of MQOM tailored to embedded devices, we demonstrate that MPCitH-based signatures can achieve competitive performance compared to other post-quantum signature families.

Our contributions go beyond a straightforward port of existing reference implementations. We introduce a set of implementation techniques specifically designed for deploying MPCitH-based schemes on microcontroller-class architectures. Most of the techniques described in this work are generic and can be applied to other MPCitH-based signature schemes as well. More precisely, our contributions are threefold:

1. We propose a highly-optimized implementation for embedded devices of the NIST candidate MQOM in all its variants (R3 and R5, Fast/Short, $\mathbb{F}_2/\mathbb{F}_{16}/\mathbb{F}_{256}$). This implementation outperforms by far the former existing implementations on such platforms, both in terms of memory footprint and running time, obtaining competitive performances compared to other post-quantum schemes. We propose the first implementation that relies on bitslicing on the parallel repetitions to optimize the running time of a MPCitH-based scheme. We also propose a new "Faster" variant that offers interesting speed and memory trade-offs.
2. While MQOM relies on correlated trees [HJ24, KLS25] to have shorter signatures, we propose the first memory-optimized implementation of the one-tree technique [BBM+24], which is an alternative approach to also reduce signature sizes used in the other NIST MPCitH-based candidates. We further provide insights to compare those two techniques from an *implementation point-of-view*.
3. We demonstrate two ways to mitigate further the running times and signature sizes by showing that streaming and precomputation (*i.e.* offline/online signature) can highly help in some contexts. For example, to decrease the latency when a storage (e.g. Flash) can store precomputed data for the signature, or when one wants to parallelize the signature verification processing small chunks of data. We show that MPCitH-based schemes outperform other post-quantum schemes in such contexts.

All the techniques presented in this work are portable across MCUs and specifically validated through experimental evaluations on a Cortex-M4 platform. The implementation used for these experiments along with further optimizations and updates are available at

`https://github.com/mqom/embedded-experiments`.

Two prior works have investigated efficient implementations of MPCitH-based signature schemes on embedded devices: [BBB+24d] studies an embedded implementation of the PERK scheme [ABB+23], while [ADE+25] focuses on FAEST [BBD+23a]. We emphasize that both of these works consider Round-1 versions of the corresponding schemes, whereas significant design changes were introduced in the second round of the NIST process. The results we provide for PERK nonetheless come from the Round-2 merged in [KPR+][1],

[1] See merged PR `https://github.com/mupq/pqm4/pull/393`.

but the PERK design has significantly changed in version 2.1, for which there is currently no implementation targeting embedded devices. To the best of our knowledge, our work presents the first memory-efficient implementation of an MPCitH-based NIST candidate following the major design updates introduced in Round 2 (with the exception of PERK, whose design update occurred in version 2.1). Furthermore, the performance achieved by our implementation significantly surpasses that reported in these earlier studies. For completeness, we must point that the [KPR+] framework also took the initiative of porting other MPCitH Round-1 and Round-2 candidates, but most of the implementations are straightforward adaptations from the designers' submitted desktop reference code (MQOM v1 being one of them), yielding very bad memory footprints and/or cycles counts.

The main goal of this article is to demonstrate that the MPCitH paradigm is not anymore (systematically) synonym to slow schemes, that the resulting schemes might be competitive (and sometimes outperform) with other families of post-quantum signature, depending on the context. It has also clear implementation advantages: no floating-point arithmetic, no Gaussian elimination, no rejection sampling, etc. Furthermore, the recent MPCitH schemes use binary fields and avoid prime based fields, which removes the need for arithmetic-boolean conversions. While the structure remains a bit heavy, especially compared to pre-quantum schemes, we believe that there are no implementation caveats.

Organization. Section 2 provides an overview of the MQOM scheme. Section 3 describes how the scheme can be optimized for memory-restricted platforms. Section 4 focuses on the one-tree technique. Section 5 illustrates how streaming and precomputation can further improve performance. Finally, Section 6 concludes the paper.

2 Preliminaries on MQOM

MQOM is a signature scheme based on the TCitH framework [FR25], which is currently an ongoing candidate to the NIST standardization process for additional post-quantum signatures. Its security relies on the hardness of solving the unstructured multivariate quadratic (MQ) problem over a base field $\mathbb{F}$. The public key is a MQ instance, while the secret key is the corresponding solution. More precisely, the public key $\mathtt{pk}$ is composed of a seed $\rho_{\mathsf{eq}} \in \{0,1\}^{2\lambda}$ that represents the MQ equations and a vector $\hat{y} \in \mathbb{K}^{\hat{m}}$ (λ is the security level in bits), while the secret key $\mathtt{sk}$ is simply a vector $x \in \mathbb{F}^n$ satisfying

$$\forall i \in [1:\hat{m}],\ x^\top \hat{A}_i x + \hat{b}_i^\top x = \hat{y}_i\ , \tag{1}$$

where $\mathbb{K}$ is a field extension of $\mathbb{F}$ of degree μ and where $\{\hat{A}_i, \hat{b}_i\}_i$ are expanded from ρ_{eq} using a pseudo-random generator ($\hat{A}_i \in \mathbb{K}^{n\times n}$ and $\hat{b}_i \in \mathbb{K}^n$ for all i). The key generation is then straightforward. It consists of sampling ρ_{eq} and x, computing $\hat{y}$ using (1), and building $\mathtt{pk}$ and $\mathtt{sk}$ as $\mathtt{pk} := (\rho_{\mathsf{eq}}, \hat{y})$ and $\mathtt{sk} := (\mathtt{pk}, x)$.

As all the (recent) MPCitH-based schemes, MQOM is obtained by converting an interactive identification scheme into a signature scheme using the Fiat-Shamir transformation [FS87]. The MQOM's Polynomial IOP (where IOP stands for "Interactive Oracle Proof") protocol, *i.e.* an interactive proof system where the prover sends oracle access to polynomials that the verifier queries at random points, is described in Figure 1. At the beginning, the prover samples two random vectors of polynomials $P_x(X) := x \cdot X + x_0 \in \left(\mathbb{K}[X]\right)^n$ and $P_u(X) := u_1 \cdot X + u_0 \in \left(\mathbb{K}[X]\right)^\eta$ such that the leading term of P_x is the secret x. Then, it commits to those polynomials as oracles. The verifier sends a random matrix $\Gamma \in \mathbb{K}^{\eta\times\hat{m}}$. The prover computes the vector polynomials $P_z : z_1 \cdot X + z_0$ and $P_\alpha : \alpha_1 \cdot X + \alpha_0$, and sends the latter. After sampling a random evaluation point from a public set $\Omega \subseteq \mathbb{K}$ for which the size is denoted N, the verifier evaluates the committed polynomials into this

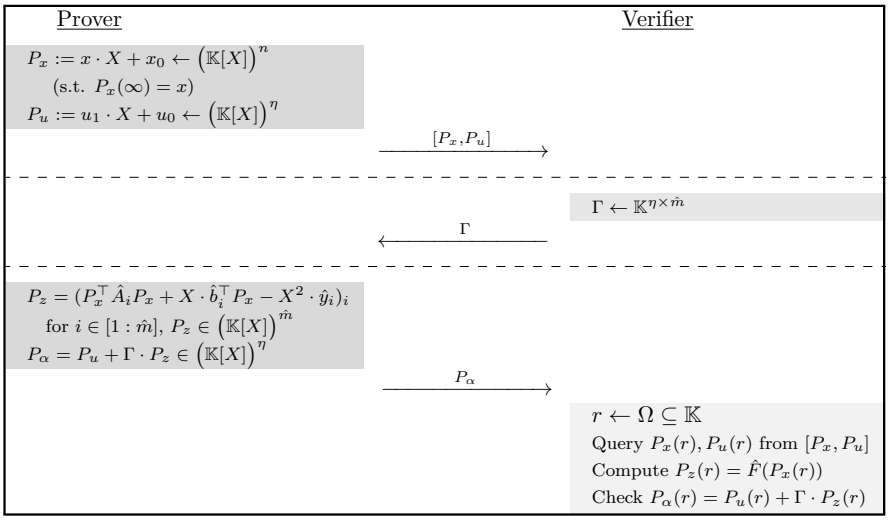


Figure 1: Polynomial IOP (PIOP) underlying MQOM, see [BBFR24].

point (using the sent oracles) and checks that the relations between P_α, P_x and P_u hold for this random point.

By default, the protocol has 5 rounds, since the verifier needs to sample two challenges[2]: the batching matrix Γ and the evaluation point r. MQOM proposes a variant R3 where there is no batching, which is conceptually equivalent to having $\Gamma \in \mathbb{K}^{\eta \times \hat{m}}$ as the identity matrix with $\eta := \hat{m}$. In that case, the corresponding protocol has only 3 rounds. The role of Γ is to decrease the size of P_α with $\eta < \hat{m}$, leading to slightly shorter signatures.

Algorithm 1 $\mathsf{Sign}(\mathtt{sk}, \mathtt{msg})$

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

1: $(\mathtt{pk}, x) \leftarrow \mathtt{sk}$
2: $(\rho_{\mathsf{eq}}, \hat{y}) \leftarrow \mathtt{pk}$
3: $(\mathsf{mseed}, \mathsf{salt}) \leftarrow \$$
4: $(\mathsf{com}_1, \mathtt{key}, x_0, u_0, u_1) \leftarrow \mathsf{BLC.Commit}(\mathsf{mseed}, \mathsf{salt}, x)$
5: $(\alpha_0, \alpha_1) \leftarrow \mathsf{ComputeAlphaLines}(\mathsf{com}_1, x_0, u_0, u_1, x, \rho_{\mathsf{eq}})$
6: $\mathsf{com}_2 \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
7: $\mathtt{hash} \leftarrow \mathsf{Hash}_4(\mathtt{pk}, \mathsf{com}_1, \mathsf{com}_2, \mathsf{Hash}_2(\mathtt{msg}))$
8: $(r, \mathtt{nonce}) \leftarrow \mathsf{SampleChallenge}(\mathtt{hash})$
9: $\mathtt{opening} \leftarrow \mathsf{BLC.Open}(\mathtt{key}, r)$
10: **return** $\mathsf{sig} := (\mathsf{salt}, \mathsf{com}_1, \mathsf{com}_2, \alpha_1, \mathtt{opening}, \mathtt{nonce})$

Algorithm 2 $\mathsf{Verif}(\mathtt{pk}, \mathtt{msg}, \mathsf{sig})$

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

1: $(\rho_{\mathsf{eq}}, \hat{y}) \leftarrow \mathtt{pk}$
2: $(\mathsf{salt}, \mathsf{com}_1, \mathsf{com}_2, \alpha_1, \mathtt{opening}, \mathtt{nonce}) \leftarrow \mathsf{sig}$
3: $\mathtt{hash} \leftarrow \mathsf{Hash}_4(\mathtt{pk}, \mathsf{com}_1, \mathsf{com}_2, \mathsf{Hash}_2(\mathtt{msg}))$
4: $(r, \mathtt{val}) \leftarrow \mathsf{DeriveChallenge}(\mathtt{hash}, \mathtt{nonce})$
5: $(\mathsf{com}'_1, x_{\mathtt{eval}}, u_{\mathtt{eval}}) \leftarrow \mathsf{BLC.Eval}(\mathsf{salt}, \mathtt{opening}, r)$
6: $\alpha_0 \leftarrow \mathsf{RecomputeAlphaLines}(\mathsf{com}_1, \alpha_1, r, x_{\mathtt{eval}}, u_{\mathtt{eval}}, \rho_{\mathsf{eq}}, \hat{y})$
7: $\mathsf{com}'_2 \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
8: **if** $(\mathsf{com}'_1 \neq \mathsf{com}_1$ **and** $\mathsf{com}'_2 \neq \mathsf{com}_2$ **and** $\mathtt{val} \neq 0)$ **then**
9: **return** False
10: **return** True

Figure 2: MQOM v2 – Signing and verification algorithms.

The signing and verification algorithms of the MQOM scheme are formally described in Algorithms 1 and 2. The signing algorithm starts by sampling a salt and a master seed. Then it samples and commits to the polynomials $P_x := x \cdot X + x_0$ and $P_u := u_1 \cdot X + u_0$ using the batch line commitment (BLC) scheme, which is the cryptographic primitive that implements the PIOP oracles in MQOM. The resulting commitment digest is denoted com_1 and the opening key (*i.e.* the key that will enable the signer to later open the commitment)

[2]The second challenge does not appear as an interaction in the PIOP formalism, but as a query instruction.

is key. It computes the polynomial $P_\alpha : \alpha_1 \cdot X + \alpha_0$ using the routine ComputeAlphaLines (that also expands the MQ equations from ρ_{eq}). Then Lines 6-8 compute the random evaluation point r as the output of hash functions. The routine SampleChallenge includes a proof-of-work (PoW) mechanism [Sta21] to computationally increase the security of the scheme, enabling it to have shorter signatures by taking smaller parameters. Finally, it calls the opening routine of the BLC scheme to reveal the evaluation $x_{\mathtt{eval}} := P_x(r)$ and $u_{\mathtt{eval}} := P_u(r)$ in a provable way. After some parsing operations, the verification algorithm starts by rederiving the evaluation point r. Then it recomputes the evaluations $x_{\mathtt{eval}}$ and $u_{\mathtt{eval}}$ while ensuring that they are consistent with the commitment digest com_1 using BLC.Eval. Finally, using α_1, $x_{\mathtt{eval}}$ and $u_{\mathtt{eval}}$, it derives P_α and checks that the latter is consistent with com_2.

Let us mention that the identification scheme described in Figure 1 should be repeated several (τ) times in parallel to be secure, so in Algorithms 1 and 2, (almost) all the involved quantities $(x_0, u_0, u_1, \alpha_0, \alpha_1)$ are length-τ arrays.

Presenting the other NIST MPCitH-based candidates. MQOM is not the only MPCitH-based scheme among the ongoing NIST candidates on the road to standardization, there are also FAEST, Mirath, PERK, RYDE and SDitH [ABC+24]. While this work focuses on optimizing MQOM, many described optimizations would also be applicable to other candidates. All the schemes rely on the protocol described in Figure 1, except that the definition of P_z differs to take into account their underlying hard problem (*e.g.* the MinRank problem for Mirath). As Mirath and RYDE also rely on the TCitH framework [FR25], the overall schemes (including the BLC definition) are close to MQOM. FAEST, PERK, and SDitH rely on the VOLEitH framework [BBD+23b], so $\mathbb{K}$ and Ω are exponentially large, implying that they do not need to rely on parallel repetitions, but the BLC scheme should include some consistency check. All the schemes (both TCitH-based and VOLEitH-based) use several GGM trees to commit to polynomials. A notable difference between MQOM and the other schemes is how those GGM trees are optimized: while MQOM relies on correlated trees [HJ24, KLS25], the other schemes use the one-tree technique [BBM+24].

3 Optimizing MQOM

In this section, we describe how we optimized MQOM for embedded devices, in order to have fast implementations with low memory footprint. While we provide a generic C code for all the MQOM implementation, we sometimes also use alternative more efficient assembly code dedicated to the ARMv7-M microarchitecture to speed up the running time on such platforms.

Before explaining the optimizations, let us explain the overall performance of the reference implementation of MQOM. The BLC scheme involves computations of τ GGM trees of N leaves, expansions of all the $\tau \cdot N$ leaf seeds using a pseudo-random generator, commitments of $\tau \cdot N$ leafs seeds and hash of those commitment digests. Except at the end of the commitment procedure where the expanded randomness is recombined to obtain the polynomials P_x and P_u, the computation relies on symmetric primitives. Therefore, the BLC computational bottleneck comes from those *symmetric primitives*. Regarding the PIOP computation (in ComputeAlphaLines and RecomputeAlphaLines), except to expand the MQ equations from the seed ρ_{eq}, the computation is mainly arithmetic and the computational bottleneck is the *matrix multiplication* involving $\{\hat{A}_i\}_i$.

Regarding the memory footprint, keeping in memory all the trees and the objects derived from their leaves (*e.g.* seed commitment digests) takes at least several hundreds to thousands of kilobytes. Moreover, the expanded MQ equations (*i.e.* the matrices $\hat{A}_i$ and the vectors $\hat{b}_i$) take also several tens to hundreds of kilobytes. This is an issue to deploy MQOM on embedded devices with restricted memory.

Studied Instances. MQOM v2 proposes 12 concrete instances per security level. To lighten the description and performance analysis, we will restrict ourselves to the instances based on the base field $\mathbb{F} := \mathbb{F}_{16}$, where $\mathbb{F}_q$ refers to the field with q elements. Introduced in the version 2.1, this base field offers "both competitive signature sizes and running times: the $\mathbb{F}_2$ variant suffers from limited computational performance, while the $\mathbb{F}_{256}$ variant incurs relatively large signature sizes" [BBFR24]. Moreover, we focus on the R5 variant, namely when the batching challenge Γ is not the identity matrix (see the definition of the R3 variant in Section 2). This variant leads to shorter signatures, and all the optimizations for it will also be applicable to the R3 variant. We recall all the parameters of the studied instances for the three NIST security levels in Table 1.

Table 1: Parameter overview for the studied MQOM instances

Instance	$\lvert\mathbb{F}\rvert/\lvert\mathbb{K}\rvert$	$n/\hat{m}$	τ	N	Private key (bytes)	Public key (bytes)	Signature (bytes)
MQOM L1 $\mathbb{F}_{16}$ fast R5	16/256	56/28	17	256	88	60	3 280
MQOM L1 $\mathbb{F}_{16}$ short R5	$16/256^2$	56/14	12	2048	88	60	2 916
MQOM L3 $\mathbb{F}_{16}$ fast R5	16/256	84/42	27	256	132	90	7 738
MQOM L3 $\mathbb{F}_{16}$ short R5	$16/256^2$	84/21	18	2048	132	90	6 496
MQOM L5 $\mathbb{F}_{16}$ fast R5	16/256	116/58	36	256	180	122	13 772
MQOM L5 $\mathbb{F}_{16}$ short R5	$16/256^2$	116/29	25	2048	180	122	12 014

Optimization Targets and Benchmarking Platforms. Most of the cycles performance and memory footprint optimizations that will be presented in this section are generic and will be applicable to any embedded platform. For our experimental setup and explorations, we have chosen to focus on the Cortex-M4 MCU family as it has become a reference architecture, acknowledged by NIST, for benchmarking PQ schemes [KPR+]. More specifically, we have used two boards: the Nucleo L4R5ZI featuring a STM32L4R5ZI with 2 MB of Flash and 640 KB of SRAM[3], and a custom board embedding a STM32F437 with 2 MB of Flash and 256 KB of SRAM. Although similar in terms of performance profile (the Cortex-M4 cores are very close), the STM32F437 has the advantage of embedding a dedicated AES hardware accelerator allowing for additional experiments. For accurate cycles measurements and a fair comparisons with the state of the art, we configure the boards at their Flash zero wait-state frequency.

3.1 Optimizing the Symmetric Primitives

As previously mentioned, MQOM heavily relies on symmetric primitives. Let us thus detail the used implementations of the symmetric primitives. MQOM both relies on

- a block cipher for pseudo-random expansion and seed commitment;
- a hash function to compress the prover's data to derive the Fiat-Shamir challenges.

The used block cipher and hash function are respectively AES-128 and SHAKE-128 for the L1 security level, and Rijndael-256-256 and SHAKE-256 for the higher security levels. Let us stress that the hash function never handles sensitive data, it only hashes public elements (*i.e.* elements that can be recovered from the signature transcript), meaning that we could rely on non constant-time implementations. However, it is not clear how to effectively leverage this property to obtain practical performance improvements. Regarding the block cipher, it mostly handles sensitive data so we need to be careful regarding the constant-time property. However, by design of MQOM, the key schedule never handles sensitive data

[3]This board is commonly used in embedded PQ benchmarking frameworks, e.g. [KPR+]. The large 640 KB of SRAM leaves room for testing many performance trade-offs.

due to the use of the Davies-Meyer construction. While the cipher evaluation might handle sensitive data at some places, there are other parts of the code where encryption is performed over public data. It is the case of the public key expansion (*i.e.* expansion of the MQ equations) or in the verification algorithm. For this reason, we rely on a set of two disjoint APIs for the block cipher: one that handles sensitive data, while the other does not. This allows to speed up the encryption in some parts of the code.

For all the symmetric primitives, we have tried to integrate state of the art efficient implementations. For SHAKE, we rely on the optimized ARMv7-M assembly from [Ado23] that takes advantage of the ARM barrel shifter. For AES-128 and Rijndael-256-256 implementations:

- The non-sensitive key schedule uses look-up tables based implementations, as these are the most efficient at reasonable additional memory cost. For AES-128 the key schedule is taken from [SS16] that uses 1024 bytes in Flash. For Rijndael-256-256 we use a portable C implementation making use of the 256 bytes S-Box in Flash. When the key schedule targets a bitslice fashion encryption (which is the case for constant-time encryption, see below), the scheduled keys must be transformed in their bitslice form: applying this transformation is more efficient than using the constant-time bitslice key schedule that is usually much slower. We also amortize this transformation cost across factorized key schedules in MQOM using dedicated `x2`, `x4` and `x8` contexts.
- The public encryption API uses [SS16] for AES-128 (with 1024 bytes table in Flash), and the classical Rijndael T-table implementation for Rijndael-256-256 using four tables of 1024 bytes in Flash. On platforms where there is no cache or prefetcher between the ARMv7-M core and the SRAM unit, the same implementations can be used as they will be constant-time[4]. For the general case behind the private encryption API, we use the assembly fixslice (bitslice where the representation optimizes `ShiftRows` by exploiting the ARM barrel shifter) AES-128 from [AP20], and we have adapted BearSSL's `ct64` bitslice AES [Por] implementation to Rijndael-256-256[5].

The bitslice implementations expect two state blocks to be packed, and hence achieve their best performance with at least `x2` encryption. Whenever encryption is performed more than once under the same keys, dedicated contexts with already bitsliced keys are used to amortize the transformation. We have implemented the variants for two times and four times encryptions, yielding various cycles versus memory trade-offs depending on how much space is used for the saved contexts: we will present the impacts of these on MQOM's BLC in the sequel. We present on Table 2 the comparative benchmarks in cycles on Cortex-M4 MCUs of the various implementations flavors (the hardware accelerated variant is discussed in a dedicated section below). The key schedule for bitslice implementations is in fact the same as the table-based one (because in MQOM this is not sensitive). The implementations for AES-128 are in optimized assembly while the implementations for Rijndael-256-256 are in portable C (and hence have room for improvement). As expected for bitslice, we see that `x1` is far less efficient than `x2` and more. Also, specifically for bitslice, we present the "amortized" variants of `x2` and `x4` where the key bitslicing transformation cost, $\approx 1\,000$ cycles for AES-128 and $\approx 3\,500$ cycles for Rijndael-256-256 per key, is in the key schedule instead of being in the encryption.

[4]This is the case for most Cortex-M4 hardware instantiations and for our benchmarking platforms, but this is not always true: for instance, the multi-core `i.MX 7` platform contains a Cortex-M4 with a 8 KB unified cache. Also, on some platforms the access latency might depend on SRAM banks switching when the volatile memory is split across multiple banks.

[5]This is a portable C implementation that could be optimized by migrating to fixslice-like assembly: this is left as future work.

STM32 Hardware AES accelerator performance The STM32F437 MCU embeds a so-called `CRYP` engine, a hardware IP external to the core (on the AHB bus) that implements AES – but not Rijndael-256-256 – allowing to accelerate MQOM L1. The measured performance in Table 2 is the combined key schedule plus encryption due to the way the accelerator works (there is a unique context), and using polling where the core actively waits for the end of IP operations. The resulting performance in cycles is approximately half of what is achieved by the assembly table-based AES-128, which seems rather disappointing[6] but is explained by the fact that the IP is external and requires some setup. The accelerator is tailored for large buffers encryption under the same key, with increased performance using DMA (Direct Memory Access) where the CPU core only programs memory transfers to and from the IP (the small setup cost being absorbed). Figure 3 compares the number of cycles taken to encrypt $0 < n < 2048$ blocks of AES using polling and DMA: they are on par under 3 blocks, but DMA becomes the clear winner for $n \geq 3$, with a increasing speedup factor of ≥ 2 beyond 100 blocks. As we will see in Section 3.2, DMA under the same key can prove particularly useful in the context of MQOM with the appropriate BLC implementation strategies.

Beyond hardware acceleration, the `CRYP` engine has also the advantage of reducing the memory footprint: no table of constants is used, and the AES scheduled keys are not stored in SRAM as they live inside the IP's internal memory blocks (scheduled keys take ≈ 700 bytes in each software AES-128 `x4` context for instance, reduced to almost zero when using the accelerator).

Table 2: Block Ciphers performance (in cycles) on the Nucleo L4R5ZI board (except for hardware based AES benchmarked on STM32F437).

		Bitslice		Table-based		Hardware
		KeySch.	Enc.	KeySch.	Enc.	KeySch. + Enc.
AES-128	`x1`	405	4 910	405	800	529
	`x2`	810	5 056	810	1 500	909
	`x2` amortized	2 751	2 879	-	-	-
	`x4`	1 620	10 037	1 620	2 900	1661
	`x4` amortized	5 405	5 660	-	-	-
Rijndael-256-256	`x1`	1 398	21 121	1 300	3 345	N/A
	`x2`	2 800	23 043	2 500	6 600	
	`x2` amortized	12 224	15 695	-	-	
	`x4`	5 600	46 488	5 334	14 000	
	`x4` amortized	24 624	32 024	-	-	

3.2 Optimizing Batched Line Commitment

The batched line commitment (BLC) scheme aims to sample and commit to the degree-1 polynomials/lines P_x and P_u for each parallel repetition of the protocol. For each repetition,

1. it expands a GGM tree of N leaves (where N is usually 256 or 2048),
2. it commits independently to those N seed leaves to obtain N commitment digests,
3. it expands all the N seeds with a pseudorandom generator and
4. it folds the expanded randomness to obtain the final polynomials/lines.

The opening procedure of the BLC scheme consists of revealing a sibling path in the GGM tree and one of the commitment digests (for each parallel repetition). To avoid any re-computation between the commitment procedure and the opening, most of the existing implementations of MPCitH-based schemes keep in memory the expanded trees and all

[6] Compared to e.g. AES-NI on `x86`.

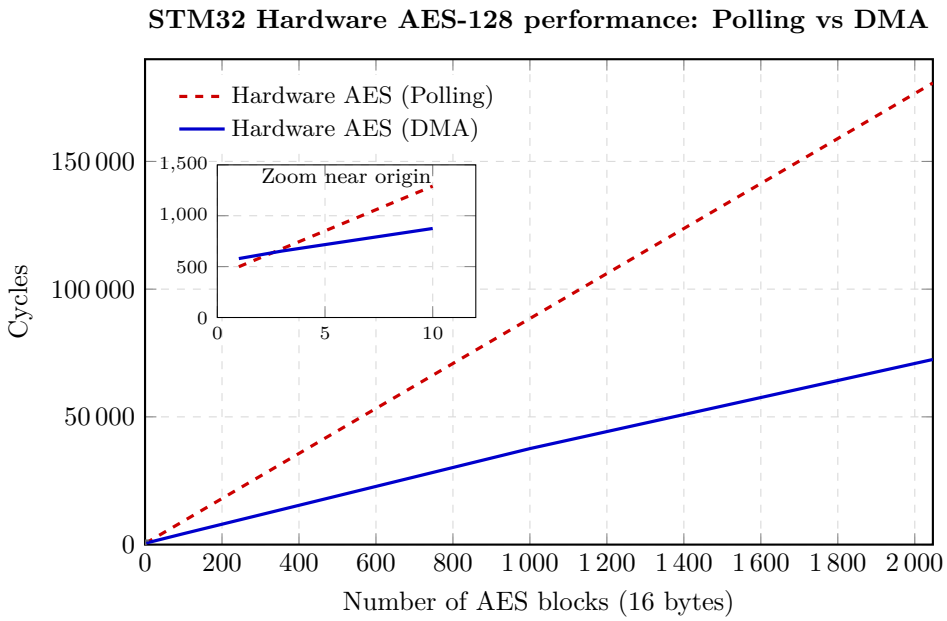


Figure 3: Comparison of hardware AES-128 (`CRYP` engine) of STM32F437 performance based on data transfer method. Polling and DMA are on par below 3 blocks, DMA becomes more efficient beyond.

the commitment digests. In that case, the memory footprint is at least of (in bits)

$$\underbrace{\tau}_{\text{nb repetitions}} \times (\underbrace{(2N-1)\cdot\lambda}_{\text{expanded tree}} + \underbrace{N\cdot(2\lambda)}_{\text{commitment digests}})$$

For MQOM, it represents 278 KB for L1 fast and 1.6 MB for L1 short. Therefore, keeping in memory all those quantities is prohibited when targeting memory-efficient implementations. Consequently, one should keep in memory only the tree roots and the opening routine would expand the sibling path of the hidden leafs in complexity $O_\lambda(\tau \cdot \log_2 N)$ and recompute their commitment digests, where $O_\lambda(\cdot)$ denotes the complexity when focusing on a given security level. This recomputation is very cheap compared to the rest of the scheme. Using T-table implementation of AES, it costs less than 200 kilocycles while the entire signing algorithm costs more than tens of megacycles (usually more than hundreds of megacycles). So if we do not need to keep all the GGM trees in memory, we can erase the expanded tree after each repetition. This is the approach described in [BBB+24d], reducing the memory footprint by a factor τ. We can go further as described in [ADE+25] (and also implemented in SDitH v2 [ABB+24] with the one-tree technique, see Section 4). Instead of expanding the entire tree in memory at the memory cost of $(2N-1)\cdot\lambda$ bits, we can adopt a tree-walk strategy to incrementally compute the leaf seeds. Using a depth-first search (DFS), we can reduce the memory footprint to $\lambda \cdot \log_2 N$ bits by simply keeping in memory the path from the root to the current leaf. Let us mention that we do not need to store the commitment digests if one uses incremental hashing. We adopt this implementation strategy for our implementation of MQOM. While it follows the approach of [ADE+25], let us highlight that the implementation differs a bit as MQOM relies on the half-tree technique to optimize the scheme. In the MQOM scheme, the node derivation depends on the Davies-Meyer mode of a block cipher (AES-128 or Rijndael-256-256). There is a domain separation at each level of the tree, implying that we can factorize the key schedule of the block cipher for a tree level, at the cost of storing the expanded cipher key. In our

implementation, we can configure how many key schedules we want to factorize (it will factorize the key schedules of the deepest nodes).

Let us also highlight that the MQOM's folding step relies on Gray code (since version 2.1), similarly to SDitH v2 [ABB+24], and so this step has a running time scaling with $N + \log_2 N$ with a constant memory footprint, while FAEST's folding relies on codewords in the lexicographic order ans so [ADE+25]'s folding has a running time scaling with $N \cdot \log_2 N$ with a constant memory footprint.

While the methodology for obtaining a memory-efficient implementation of the GGM tree/BLC essentially relies on a depth-first tree traversal combined with incremental processing of the generated leaves, a key question remains: how can one take advantage of the speed-ups offered by implementations that support parallel encryption? For instance, the bitsliced AES/Rijndael implementation described in the previous section achieves optimal performance when executing two encryptions simultaneously. However, a depth-first tree traversal is inherently sequential and not naturally parallelizable. In the following, we explore three strategies to address this limitation.

Parallel walks. In an earlier version of this work, we explored generalizing our tree implementation to process multiple trees simultaneously, thereby enabling the concurrent computation of multiple BLC instances. Streaming two trees in parallel requires a memory footprint of $2 \cdot \lambda \cdot \log_2 N$ bits (instead of $\lambda \cdot \log_2 N$), but allows us to benefit from the `x2` speed-up available in some implementations (see Table 2). We also implemented a `x4` variant to exploit platforms supporting higher parallelism (e.g., AES-NI on `x86`). However, beyond the increased memory for the tree traversal itself, this approach also requires duplicating other components of the BLC computation, such as maintaining multiple incremental hash contexts. As a result, the overall memory overhead is significant, making this approach unsuitable for very constrained environments. This explains why the memory-efficient profiles in the previous version of this work incurred substantial computational penalties, as we did not leverage this speed-up.

Advanced walk within a tree. Instead of processing multiple trees, one can batch node computations within a single tree. However, due to the half-tree technique (where the left child requires a block cipher call and the right child is obtained via an XOR with the parent), leveraging `x2` encryption requires computing cousin nodes simultaneously. This leads to a more complex traversal strategy and increased memory usage. The situation becomes even more intricate with `x4` encryption, which requires computing four cousin nodes sharing a common great-grandparent. One possible solution is to define an alternative tree structure: each node in this tree corresponds to $2p$ sibling nodes in the original tree (sharing a common ancestor at some depth[7]). This enables deriving nodes using p parallel encryptions. However, this comes at the cost of a larger memory footprint, namely $(2p) \cdot \lambda \cdot \log_2(N/(2p))$, compared to $p \cdot \lambda \cdot \log_2 N$, which reduces a bit its practicality in memory-constrained settings.

In-place subtree computation. A third approach consists in computing subtrees in an in-place manner. Given a parameter β, we perform a tree traversal down to depth $\log_2 N - \log_2 \beta$, and then fully expand the corresponding subtree of size β. Since internal nodes of the subtree do not need to be stored, this expansion can be done in-place over the leaves storage space. This corresponds to a DFS traversal for the $\log_2 \beta$ first levels combined with a BFS (Breadth-First Search) traversal for the $\log_2 N - \log_2 \beta$ next levels. This approach offers several advantages. First, it enables efficient batching of computations at deeper levels of the tree, where most of the work lies. Second, all nodes at a given level of the subtree use the same cipher key, which allows efficient batching on hardware accelerators and reduces overhead in software

[7] If $p = 1$, they share the same parent; if $p = 2$, they share the same grandparent, etc.

implementations (*e.g.*, fewer key schedule loads). This idea can also be extended to seed commitment and expansion: instead of processing one seed at a time, we process β seeds in parallel, computing their expanded blocks layer by layer (*e.g.* first deriving the first λ-bit chunk of the commitment for all those leaves, then deriving the second one), still exploiting key schedules factorization. Overall, this approach provides a tunable trade-off between memory and performance, allowing adaptation to different constraints. Notably, for sufficiently large β, storing scheduled keys may even become unnecessary due to amortization.

After testing the three approaches, our final implementation adopts the in-place subtree computation strategy combined with a basic DFS traversal for the upper levels of the tree as this provides the best results for all the memory versus speed trade-offs. While it would be possible to use the advanced traversal for the upper levels, the resulting efficiency gains would be marginal, since these levels do not constitute a computational bottleneck, and would come at the cost of increased implementation complexity.

Finally, let us stress that all the mentioned options can be applied to BLC.Eval, which has a structure similar to BLC.Commit. Further optimizations –not detailed here for brevity– leave room for fine grained trade-offs between memory and cycles: the curious reader can refer to the implementation.

Remark 1. Although MQOM embedded implementations end up using the in-place subtree computation strategy that computes multiple block encryptions under the same key, we stress that `x2`, `x4` (and so on) variants could be useful in the context of GGM trees for MPCitH schemes depending on the keys and cipher usage. The current frameworks integrating PQ cryptography [KPR+, KSSW22, SM16] only offer AES APIs that are limited, at the best, to parallelized encryption of multiple chunks under the same key. They also strongly discourage the usage of local implementations provided by the submitted schemes. Providing such APIs might help the exploration of MPCitH candidates performance trade-offs.

3.3 Optimizing PIOP

While the mechanism of the BLC scheme is common to most MPCitH-based schemes (including VOLEitH-based ones), it is not the case for the PIOP part. In MQOM, the PIOP suffers from a major drawback: it handles large matrices $\{\hat{A}_i\}_i \in (\mathbb{K}^{n\times n})^{\hat{m}}$, leading to high memory footprint and slow running times. The critical point consists of computing $\hat{A}_i P_x$ for all i (*c.f.* Figure 1), for each parallel repetition. We can write $P_x(X)$ as $x \cdot X + x_0^{(e)}$, where x is common to all repetitions and $x_0^{(e)}$ is different for each repetition ($e \in \{1, \ldots, \tau\}$). It means that we need to compute $\{\hat{A}_i x_0^{(e)}\}_i$ τ times and $\{\hat{A}_i x\}_i$ once, where $\hat{A}_i \in \mathbb{K}^{n\times n}$, $x_0^{(e)} \in \mathbb{K}^n$ and $x \in \mathbb{F}^n$. We recall the matrix parameters in Table 3, together with their memory footprint. Let us emphasize that the field $\mathbb{K}$ is either $\mathbb{F}_{256}$ in the Rijndael representation or its field extension $\mathbb{F}_{256^2}$. It implies that the base arithmetic in MQOM is over the field $\mathbb{F}_{256}$, even when the secret lives in $\mathbb{F}_2$ or $\mathbb{F}_{16}$.

Table 3: Parameters and memory footprint of the MQ matrices

Instance	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	n	$\hat{m}$	Memory cost
L1 $\mathbb{F}_2$ fast/short	2	256 / 256^2	160	20 / 10	258 KB
L1 $\mathbb{F}_{16}$ fast/short	16	256 / 256^2	56	28 / 14	45 KB
L1 $\mathbb{F}_{256}$ fast/short	256	256 / 256^2	48	48 / 24	56 KB
L5 $\mathbb{F}_2$ fast/short	2	256 / 256^2	320	40 / 20	2 054 KB
L5 $\mathbb{F}_{16}$ fast/short	16	256 / 256^2	116	58 / 29	394 KB
L5 $\mathbb{F}_{256}$ fast/short	256	256 / 256^2	96	96 / 48	447 KB

On modern `x86` platforms, we can use GFNI instructions to have an efficient implemen-

tation for the matrix multiplication. Even without GFNI, we can use AVX2 or AVX-512 to have quite efficient implementations. For such platforms, the memory footprint remains contained. However, implementing this is more tricky when focusing on embedded devices. Although the Nucleo L4R5ZI embeds 640 KB of SRAM, it is among the largest existing MCUs. A more reasonable target for signature algorithms implementations would be less than 100 KB on medium range devices, and less than 10 KB for constrained ones.

First of all, to mitigate the memory footprint of those matrices $\{\hat{A}_i\}_i$, we implemented an on-the-fly matrix multiplication. We expand $\{A_i\}$ row by row (one matrix at a time). After expanding a row $\texttt{row} \in \mathbb{K}^n$, we compute $\texttt{row}^\top x$ and $\texttt{row}^\top x_0^{(e)}$ for all e. It forces us to compute the matrix multiplication for the τ repetitions at the same time, otherwise we would need to expand the matrices τ times.

This on-the-fly computation totally mitigates the memory cost of $\{A_i\}_i$. It remains to mitigate the computational cost, which boils down to finding the most efficient way of computing multiplications in $\mathbb{F}_{256}$. We first implemented three variants for this multiplication: using log-exp tables, using a 65 KB table, and using a circuit-based implementation. Let us stress that the two first approaches are constant-time only when there is no cache system on the SRAM (assuming tables are stored in SRAM)[8]. We give the performance we obtain for each security level in Table 4. We only gave the benchmarks for the MQOM instances targeting fast running times, as the matrix multiplication does not represent a computational bottleneck in the other trade-off "short" (not because the multiplication would be cheaper, but because other parts of the scheme would be much more expensive).

Table 4: Benchmarks for Matrix Multiplication on Cortex-M4 (Nucleo L4R5ZI STM32 board). Figures are in Megacycles (Mc). For the bitslice variants, the included input and output transformations costs are provided between parenthesis.

	$\mathbb{F}_{16}$ Fast L1 $n = 56/\hat{m} = 28$ $\tau = 17$	$\mathbb{F}_{16}$ Fast L3 $n = 84/\hat{m} = 42$ $\tau = 27$	$\mathbb{F}_{16}$ Fast L5 $n = 116/\hat{m} = 58$ $\tau = 36$
$\mathbb{F}_{256}$ Log/Exp tables*	15.3	80.3	277
Full $\mathbb{F}_{256}$ mult table†	10.7	51.3	176
Basic circuit	64.2	348	1213.7
SWAR 32 bits	21.3	104.8	353.4
Bitslice	20.0 (0.1)	66.1 (0.3)	338.9 (0.5)
Bitslice with jump	11.9 (0.1)	39.2 (0.3)	199.0 (0.5)
Bitslice composite	15.8 (0.1)	52.2 (0.3)	278.1 (0.5)
Bitslice composite with jump	**9.9** (0.1)	**32.3** (0.3)	**160.5** (0.5)

* Log/Exp tables use ≈ 512 bytes in SRAM.

† $\mathbb{F}_{256}$ full multiplication table uses ≈ 65 Kilobytes in SRAM.

The first idea to optimize the matrix multiplication consists in following the same approach than on larger platforms: vectorizing. We use the SWAR technique (SIMD Within A Register) to vectorize 4-ways operations within 32-bit registers. It enables us to speed-up the circuit-based implementation.

The second idea would be to rely on bitslicing. Unfortunately, using bitslicing in the same way that we used vectorizing would not be interesting, as it would require to transform the matrices $\{\hat{A}_i\}_i$ in a bitsliced representation, which would be costly as these matrices are large. An ideal solution would consist in expanding $\{\hat{A}_i\}_i$ directly in bitsliced representation, but this deviates from the MQOM specifications (and this would also prevent the use of the GFNI instructions on `x86` platforms). Moreover, while the cost in bits of those matrices is large, the number of non-zero coefficients is relatively small. For example, in L1 $\mathbb{F}_{16}$, the i-th row has 56 coefficients but in practice, has i only non-zero

[8] In most Cortex-M4 instantiations, constant-time access to Flash is not granted because of additional prefetchers.

coefficients as $\{\hat{A}_i\}_i$ are triangular (with $i \in \{1, \ldots, 56\}$). For all those reasons, we do not explore the options requiring to have a bitsliced representation of $\{\hat{A}_i\}_i$. Instead we bitsliced our implementation across the τ parallel repetitions. Therefore the base operation we need to implement is the hybrid $\mathbb{F}_{256}$ multiplication between a field element a and 32 bitsliced field elements split across eight 32-bit registers $\{\texttt{b[0:31]}_j\}_{j\in\{1,\ldots,8\}}$. To compute $c^{(e)} := a \cdot b^{(e)}$ for all $e \in \{0 \ldots, 31\}$ with $a, b^{(e)} \in \mathbb{F}_{256}$, we implement the interface

$$\{\texttt{c[0:31]}_j\}_{j\in[1,8]} = \texttt{bitsliced_F256_mult}(a, \{\texttt{b[0:31]}_j\}_{j\in[1,8]})$$

where $\texttt{b[0:31]}_j$ (resp. $\texttt{c[0:31]}_j$) denotes the j^{th} bit of the 32 field elements $(b^{(0)}, \ldots, b^{(31)})$ (resp. $(c^{(0)}, \ldots, c^{(31)})$). Using this operation, we can easily build a matrix multiplication between a common matrix and 32 (bitsliced) vectors. Specifically, to compute $t^{(e)} := \hat{A}_i x^{(e)}$ for all $e \in \{0 \ldots, 31\}$ with $\hat{A}_i \in \mathbb{F}_{256}^{n\times n}$ and $x^{(e)} \in \mathbb{F}_{256}^{n}$, we compute, for all $k \in [1, n]$,

$$\{\texttt{t}[k]\texttt{[0:31]}_j\}_{j\in[1,8]} = \sum_{\ell=1}^{n} \texttt{bitsliced_F256_mult}\left((\hat{A}_i)_{k,\ell}, \{\texttt{x}_0[\ell]\texttt{[0:31]}_j\}_{j\in[1,8]}\right)$$

where $\texttt{x}_0[\ell]\texttt{[0:31]}_j$ and $\texttt{t}[k]\texttt{[0:31]}_j$ are the j^{th} bit of the 32 field elements $((x_0^{(0)})_\ell, \ldots, (x_0^{(31)})_\ell)$ and $(t_k^{(0)}, \ldots, t_k^{(31)})$, respectively. The advantage of this approach is that the cost of transforming field elements into a bitsliced representation is limited to the vectors, and so it is cheap. Let us also highlight that the matrices $\{\hat{A}_i\}_i$ are public and so we do not need to have a constant-time implementation with respect to matrix coefficients: we only aim at constant-time implementation with regards to the secret data, *i.e.* our vectors in this case. We propose four bitsliced implementation strategies:

- The basic one implements the bitsliced circuit-based $\mathbb{F}_{256}$ multiplication assuming that $\mathbb{F}_{256}$ elements are represented as degree-7 polynomials of $\mathbb{F}_2[X]$, which is the native representation in MQOM. The pseudo-code of this strategy is formally described in Algorithm 3.
- The second implementation uses conditions to process the matrix coefficients to rely on a different code depending on each coefficient bit (thus there are 8 conditions per multiplication, these conditions are not sensitive as they depend on public data). The pseudo-code of this strategy is formally described in Algorithm 4.
- The third code implements the bitsliced circuit-based $\mathbb{F}_{256}$ multiplication assuming that $\mathbb{F}_{256}$ is built as a tower extension $\mathbb{F}_2 \to \mathbb{F}_4 \to \mathbb{F}_{16} \to \mathbb{F}_{256}$. In this implementation, the bitsliced transformation includes a change of the field representation. The pseudo-code of this strategy is formally described in Algorithm 5. Moreover, the matrix coefficients are adapted using a simple 256-byte look-up table.
- The fourth implementation follows the idea of the third one, except that the $\mathbb{F}_{16}$ multiplication distinguishes between the 16 possible options using conditions/jumps. The pseudo-code of this strategy is formally described in Algorithm 6.

Let us observe that this bitslicing is not always optimal depending on τ. For the instance "L1 fast", τ is equal to 17, so we "waste" 15 positions when working on 32-bit registers. Despite this, we observe that we achieve very competitive running times. Using bitslice, we have better performance than using a 65 KB table, with low memory footprint and while being constant-time with regards to secret data (independently on the usage of cache system on the SRAM). Even if we "waste" some positions, we can mitigate this observation by the following fact. We can use a bitslicing position to compute $\hat{A}_i x$ simultaneous to $\{\hat{A}_i x_0^{(e)}\}_e$, embedding $x \in \mathbb{F}^n$ in $\mathbb{K}^n$. This might appear to be overkill, but this comes for free using the leftover bits in the register. Additional unused bits can prove useful in future works such as efficient implementations resilient against side-channel and fault attacks.

Algorithm 3 Without jumps

Input: a field element $a := \sum_{i=0}^{7} a_i \cdot \delta^j$ and 32 bitsliced field elements $\{\texttt{b[:]}_j\}_{j\in[1,8]}$
Output: 32 bitsliced field elements $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

1: **for** $i = 7$ **down to** 0 **do**
2: $\quad \texttt{lead} \leftarrow \texttt{c[:]}_7$
3: $\quad$ **for** $j = 7$ **down to** 0 **do**
4: $\quad\quad \texttt{c[:]}_j \leftarrow \texttt{c[:]}_{j-1} \oplus (p_j \cdot \texttt{lead}) \oplus (a_i \cdot \texttt{b[:]}_j)$
5: **return** $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

Algorithm 4 With jumps

Input: a field element $a := \sum_{i=0}^{7} a_i \cdot \delta^j$ and 32 bitsliced field elements $\{\texttt{b[:]}_j\}_{j\in[1,8]}$
Output: 32 bitsliced field elements $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

1: **for** $i = 7$ **down to** 0 **do**
2: $\quad \texttt{lead} \leftarrow \texttt{c[:]}_7$
3: $\quad$ **if** $a_i = 1$ **then**
4: $\quad\quad$ **for** $j = 7$ **down to** 0 **do**
5: $\quad\quad\quad \texttt{c[:]}_j \leftarrow \texttt{c[:]}_{j-1} \oplus (p_j \cdot \texttt{lead}) \oplus \texttt{b[:]}_j$
6: $\quad$ **else**
7: $\quad\quad$ **for** $j = 7$ **down to** 0 **do**
8: $\quad\quad\quad \texttt{c[:]}_j \leftarrow \texttt{c[:]}_{j-1} \oplus (p_j \cdot \texttt{lead})$
9: **return** $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

Figure 4: Bitsliced $\mathbb{F}_{256}$ multiplication `bitsliced_F256_mult`, assuming that $\mathbb{F}_{256}$ elements are represented as degree-7 polynomials of $\mathbb{F}_2[X]$, *i.e.* $\mathbb{F}_{256} \cong \mathbb{F}_2[\delta]/(\delta^8 \oplus \sum_{i=0}^{7} p_i \cdot \delta^i)$. We use $\overline{p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0} = \texttt{0x1B}$ (Rijndael field) in practice. By convention, $\texttt{c[:]}_{-1} := 0$.

Algorithm 5 Without jumps

Input: a field element $a := a_1 \cdot \delta + a_0$ and 32 bitsliced field elements $\{\texttt{b[:]}_j\}_{j\in[1,8]}$
Output: 32 bitsliced field elements $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

1: $s_a \leftarrow a_1 \oplus a_0$
2: $\{\texttt{s}_b\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \{\texttt{b[:]}_j \oplus \texttt{b[:]}_{4+j}\}_{j\in[1,4]}$
3: $\{\texttt{r}_0\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_mult}(s_a, \{\texttt{s}_b\texttt{[:]}_j\}_{j\in[1,4]})$
4: $\{\texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_mult}(a_0, \{\texttt{b[:]}_j\}_{j\in[1,4]})$
5: $\{\texttt{r}_2\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_mult}(a_1, \{\texttt{b[:]}_j\}_{j\in[5,8]})$
6: $\{\texttt{v[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_scale}_p(\{\texttt{r}_2\texttt{[:]}_j\}_{j\in[1,4]})$
7: $\{\texttt{c[:]}_j\}_{j\in[1,4]} \leftarrow \{\texttt{v[:]}_j \oplus \texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]}$
8: $\{\texttt{c[:]}_j\}_{j\in[5,8]} \leftarrow \{\texttt{r}_0\texttt{[:]}_j \oplus \texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]}$
9: **return** $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

Algorithm 6 With jumps

Input: a field element $a := a_1 \cdot \delta + a_0$ and 32 bitsliced field elements $\{\texttt{b[:]}_j\}_{j\in[1,8]}$
Output: 32 bitsliced field elements $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

1: $s_a \leftarrow a_1 \oplus a_0$
2: $\{\texttt{s}_b\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \{\texttt{b[:]}_j \oplus \texttt{b[:]}_{4+j}\}_{j\in[1,4]}$
3: $\{\texttt{r}_0\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_scale}_{s_a}(\{\texttt{s}_b\texttt{[:]}_j\}_{j\in[1,4]})$
4: $\{\texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_scale}_{a_0}(\{\texttt{b[:]}_j\}_{j\in[1,4]})$
5: $\{\texttt{r}_2\texttt{[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_scale}_{a_1}(\{\texttt{b[:]}_j\}_{j\in[5,8]})$
6: $\{\texttt{v[:]}_j\}_{j\in[1,4]} \leftarrow \texttt{bs_F16_scale}_p(\{\texttt{r}_2\texttt{[:]}_j\}_{j\in[1,4]})$
7: $\{\texttt{c[:]}_j\}_{j\in[1,4]} \leftarrow \{\texttt{v[:]}_j \oplus \texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]}$
8: $\{\texttt{c[:]}_j\}_{j\in[5,8]} \leftarrow \{\texttt{r}_0\texttt{[:]}_j \oplus \texttt{r}_1\texttt{[:]}_j\}_{j\in[1,4]}$
9: **return** $\{\texttt{c[:]}_j\}_{j\in[1,8]}$

Figure 5: Bitsliced $\mathbb{F}_{256}$ multiplication `bitsliced_F256_mult`, assuming that $\mathbb{F}_{256}$ is built as a degree-2 extension of $\mathbb{F}_{16}$, *i.e.* $\mathbb{F}_{256} \cong \mathbb{F}_{16}[\delta]/(\delta^2 \oplus \delta \oplus p)$. `bs_F16_mult` is the $\mathbb{F}_{16}$ multiplication with the same API than `bitsliced_F256_mult`, while $\texttt{bs_F16_scale}_\beta$ is the $\mathbb{F}_{16}$ multiplication optimized for when one operand is β. The jumps that depend on public input data a for Algorithm 6 are related to the fact that $\texttt{bs_F16_scale}_{\{s_a,a_0,a_1\}}$ use a different code depending on the scalar.

Remark 2. In the key generation, such a matrix multiplication is also present to derive $\hat{y}$ from the secret x. However, in this context, it would be inefficient to rely on repetition-based bitslicing since this matrix multiplication is performed only once. Therefore, in this context, we use the constant-time SWAR implementation by default (or LUT based ones in contexts where SRAM access is constant-time, or non-sensitive usage with regards to SCA).

Remark 3. Although we can see on Table 4 that bitslicing composite with jump is the clear winner on our benchmarking platforms, we have chosen to include the other variants in our implementation of MQOM as this might not be the case on other targets. Indeed, the usage of conditional jumps or fields representations LUTs might produce different performance profiles depending on the presence of a data or instruction cache, or a branch prediction unit.

While the bitslicing approach is clearly the faster one in the context of MQOM, it comes with a drawback: as we process all the parallel repetitions *at the same time*, we need

to keep the data of all those repetitions in memory. Therefore, this prevents us to achieve very small memory footprints (*e.g.* smaller than 10 KB). Consequently, to achieve such highly optimized memory footprints, we abandon bitslicing in favor of using the SWAR technique (when targeting constant-time implementations with regards to secret data) and proceed repetition by repetition. Since this would require expanding the MQ equations τ times, our implementation is tuned to handle the repetitions by groups of `nb_para_reps` repetitions, where `nb_para_reps` is a flexible parameter. The choice of this parameter enables various memory versus speed trade-offs, as we need to perform $\lceil \frac{\tau}{\texttt{nb_para_reps}} \rceil$ MQ expansions while keeping in memory the data for the `nb_para_reps` repetitions.

Comparison with the state of the art of multivariate field arithmetic For portability across MCUs, all our field arithmetic code is in C: being on small binary fields $\mathbb{F}_{256}$ and $\mathbb{F}_{256^2}$, operations boil down to simple `xor`, shifts and/or LUTs with implicit reduction on primitive types `uint8_t` and `uint16_t`, which are decently optimized by the compiler. The literature about small fields arithmetic implementations in embedded platforms has been very active in the recent years, specifically with multivariate based candidates such as Rainbow, UOV and MAYO. Rainbow [CKY21] discusses optimized assembly ARMv7E-M Cortex-M4 strategies for constant time field multiplication over $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ based on tower field representation built on $\mathbb{F}_4$ and $\mathbb{F}_2$, yielding an average of 1 clock cycle per field element multiplication in bitsliced form for $\mathbb{F}_{16}$ (they do not provide an implementation for $\mathbb{F}_{256}$). UOV [BCH+23] extends these optimizations further for multiplying 4 fields elements by 1 (used for matrix column-fashion multiplication) by exploiting pre-computations: they provide a $\mathbb{F}_{256}$ multiply-accumulate assembly primitive that can achieve an average of 7.6 cycles per element when reusing the pre-computations (without accounting for registers/memory storage), as well as a bitslice mode achieving 5 cycles per element (without accounting for bitslice transformations and registers/memory storage, and using floating point registers to limit registers pressure). From Table 4, our approach of bitslicing across the τ repetitions provides an average of 6.6 cycles per element for $\tau = 17$, 4 cycles for $\tau = 27$, and 5.7 cycles for $\tau = 36$ (accounting for the bitslice transformation plus all the memory loads and stores). This outperforms the previous results when considering that the UOV bitslice approach with 5 cycles per element is not efficiently usable for MQOM's matrix multiplication since the public matrices must either be kept in bitsliced form for the τ repetitions using large memory, or transformed again at re-computation adding many more cycles and memory transfers. We also stress that these results could certainly be optimized using dedicated ARMv7E-M assembly, which is left as future work.

For completeness, [BCC+24b] proposes an optimized implementation of the signature scheme MAYO [BCC+24a] for AVX2 and Cortex-M4: they suggest to use the Method of the Four Russians (M4R) [ADKF70, AHU74] combined with nibble-slicing, where elements of $\mathbb{F}_{16}$ are stored packed in half-bytes. When trying to adapt such a method to MQOM with matrices over $\mathbb{F}_{256}$ using stripes $t \geq 2$, we would have to pre-compute multiple tables of 65 KB for the linear combinations of the rows of the right-hand matrix (that will be indexed by the elements of the left-hand public matrix in the product), while gaining a factor at most t (without the pre-computations) over the full $\mathbb{F}_{256}$ multiplication table based matrix product from Table 4. The induced large memory footprint is not compatible with constrained devices, explaining why M4R is ruled out.

3.4 Optimizing Misc

Proof of Work. The calls to the BLC and PIOP routines represent most of the signing computation. The rest of the signing algorithm includes some parsing and small hashing operations. The only non-trivial operation is the call to the routine SampleChallenge. The latter aims to derive the evaluation point r in which the committed polynomials will be opened into. This operation integrates some proof-of-work of w bits (where w is a scheme

parameter). It means that the routine SampleChallenge will perform approximatively 2^w hash operations before finding the evaluation point r. While we can use `x4` hash operations to speed up the computation on platforms with large SIMD vectorization (e.g. `x86` with SSE, ARM with NEON, PowerPC with AltiVec, etc.), it is not possible on embedded devices where the ISA is usually restricted. Therefore, depending on the weight of the proof-of-work, SampleChallenge might be computationally expensive. Its average computational cost naturally scales linearly with 2^w. In practice, on the Nucleo L4R5ZI, each PoW hash operation costs $\approx$ 11-12 kilocycles. As a consequence, if we have a proof of work of at least 7 bits, the average running time of SampleChallenge will be larger than one megacycle. Unless relying on a highly optimized implementation of Keccak, it is not clear how we could mitigate the computational impact of the proof of work (see Section 5.2 for further details).

Per-repetition Computation. We can observe that, in the signing algorithm, we need to compute BLC.Commit for all the parallel repetitions before running ComputeAlphaLines for any repetition. It comes from the fact that we need to know com_1 (as it is used to derive Γ) to run ComputeAlphaLines, and that com_1 is the hash digest of the committed polynomials of *all* the parallel repetitions. Therefore, it forces us to store P_x and P_u for all the repetitions between BLC.Commit and ComputeAlphaLines. This is not the case for the verification algorithm as com_1 is included in the signature transcript. When targeting verification with low memory, we can compute repetition per repetition (or by group of repetitions).

3.5 Experimental Results

Benchmarking Metrics. The main metrics that are of interest on embedded platforms are the timing performance (usually measured in cycles), the footprint in volatile memory (SRAM – which includes stack, heap and global/static variables), and the footprint in non-volatile memory (Flash – which includes the code as well as the constants, `const` in C). In the current context, we will mostly focus on cycles and SRAM footprint as they are the most critical metrics (current MCUs, even low-end, tend to have much more Flash than SRAM). We however provide an overview of the Flash usage for the four optimization profiles and the "Faster" instance that will be introduced and detailed hereafter. Using `arm-none-eabi-size` to measure the `.text` and `.rodata` sections size, we have the following for MQOM L1 $\mathbb{F}_{16}$ R5 Faster[9]:

- The LUT profile uses $\approx$ 60 KB in Flash.
- The Balanced profile uses $\approx$ 65 KB in Flash.
- The Memory profile uses $\approx$ 43 KB in Flash.
- The Hardware profile uses $\approx$ 51 KB in Flash.

Benchmarking Methodology. As previously explained, our benchmarks are performed on the Nucleo L4R5ZI board at 16 MHz and the custom STM32F437 board at 21 MHz (for the figures concerning the hardware AES acceleration) achieving a zero wait-state Flash. Our approach to counting cycles and memory usage is very similar to [KPR+] with some adaptations, hence the usage of a custom benchmarking framework. Counting the cycles uses the DWT unit of the Cortex-M4, with overflow detection using the lower resolution `SysTick` for long measurements (since the DWT register is only 32 bits and can rapidly wrap around). Counting the SRAM usage uses three probes:

[9]The provided sizes naturally exclude the `libc` contributions to the resulting firmware as we are only interested in the MQOM code.

- A probe for measuring stack usage with stack spraying (same as [KPR+]). We also double-check this usage with the static analysis tools provided by `gcc` through the `'-fstack-usage -fdump-ipa-cgraph'` that generates per-function local stack usage and the call graph (fusing the two information allows to compute cumulative worse case stack usage).
- A probe for measuring the global and static variables: this is done with a dedicated `section` in the ELF and the help of the linking `ld` script (inspired from [KPR+]).
- A probe for measuring the dynamic allocation usage: this is performed by instrumenting hooks of `malloc` and `free`, so that we have peak (maximum) allocation usage. This substantially diverges from [KPR+] approach, since their framework avoids dynamic allocations, possibly by using VLA (Variable Length Arrays) on the stack. VLAs are usually considered to lack portability and produce more fragile code, and should be avoided. Although dynamic allocation is often regarded "bad practice" in embedded systems, we point that for MQOM its usage is limited to caching a small amount of objects (*e.g.* the pseudo-random expansion output, etc.), while a purely static allocation with optimal lifetime would require more complex code.

The reported SRAM memory footprint in our results is the *sum of these three probes*: this represents a conservative number as it is an upper bound of the real usage since the peak dynamic allocation might not co-occur with the deepest stack usage. Finally, the code is compiled with `arm-none-eabi-gcc` toolchain 14.2.1 with the `-O3` flag, as well as `'-fdata-sections -ffunction-sections'` and the `-Wl,-gc-sections` linking flag to garbage collect unused data and functions in the produced ELF file.

Implementations Trade-offs and Results. In the previous sections, we described several optimizations. Our code also includes other minor code optimizations, such as aliasing functions local variables stack space depending on their lifetime[10]. These optimizations can be independently activated or deactivated thanks to C preprocessing flags. Therefore, our code is highly versatile with advanced configuration of the options targeting the preferred memory-speed trade-off for a given MCU. Our benchmark focuses on four profiles: LUT, Balanced, Memory and Hardware. All these profiles are *constant-time with regards to secret data*, except for the LUT based one for which it is the case only on platforms where there is no cache-like system between the core and the SRAM. We give in Table 5 the obtained running times for instances targeting the L1 security level. The instances "Fast" and "Short" come from the MQOM specifications. The instance "Faster" is an alternative one in which the GGM trees have $N := 64$ leaves ($\tau := 25$). For all the profiles we use the in-place subtree computation strategy described in Section 3.2 for the BLC optimization, with various β values that are precised in the following detailed configurations:

- **LUT.** This profile allows the use of look-up tables for sensitive data access in SRAM. It will use bitslicing for matrix multiplication, as it is always better in speed and size than LUT based $\mathbb{F}_{256}$ multiplication, and table-based block cipher. For the BLC, we use $\beta = 5$.
- **Balanced.** This profile seeks an interesting balanced trade-off between memory and speed. It will use bitslicing for multiplication and fixsliced block cipher. We fix $\beta = 5$ for the Faster and Fast variants, and $\beta = 7$ for the Short one.
- **Memory.** This profile seeks to minimize the memory footprint, at the cost of slower algorithms. It will use SWAR for matrix multiplication and fixsliced block cipher. All the existing code options are set such as the memory footprint is minimized (*e.g.* minimizing the number of parallel repetitions for PIOP and BLC, limiting the key

[10] An example would be the aliasing of α_0 and α_1 with u_0 and u_1 in the signature Algorithm 1: they share the same size, and the u_i are not used after computing the α_i.

schedule factorization, etc.). Specifically for the BLC, we use $\beta = 4$. It is interesting to notice that the memory footprint of the signature and verification is as small as *the worse case memory usage* between PIOP and BLC, meaning that there is no need to squeeze one more than the other, which boils down to finding a sweet spot to limit the speed loss.

- **Hardware.** This profile has access to an AES hardware accelerator to speed up the computation of the block cipher. It will use bitslicing for the matrix multiplication. For the BLC, $\beta = 5$ for the Faster and Fast variants, while $\beta = 7$ for the Short one to maximize batching with DMA (see Section 3.1) on the large tree while keeping a decent memory usage.

We give in Table 6 the obtained running times for instances targeting the L5 security level. With the profile targeting stringent memory usage, we achieve a memory footprint below 7 KB for L1, which results in an overall footprint below 11 KB when including the key and signature buffers[11]. The balanced profile reduces running times by a factor of 1.5–2 while maintaining controlled memory usage: below 15 KB in most L1 configurations. Finally, the table-based and hardware-assisted profiles further reduce running times, but rely on additional hardware assumptions, namely assumptions on constant-time SRAM accesses for lookup tables, or the presence of an AES hardware accelerator for the hardware-assisted profile. Let us emphasize that a wide range of intermediate profiles can be obtained between the balanced and memory-optimized configurations using the optimizations toggles flexibility. Finally, these optimizations scale well with the security level and fit comfortably in the board SRAM: for L5, only 19 KB are necessary (34 KB with the key and signature buffers) using the Memory profile, and below 50 KB using the Balanced one – detailed results for L3 are not provided here for conciseness, please refer to the provided code.

Detailed Benchmarks. Figure 6 presents detailed benchmarks of our implementation for the L1 parameter set, breaking down the running times of the different components of the scheme.

For the trade-off "Short", we clearly observe that symmetric primitives constitute the main computational bottleneck. In particular, the most expensive operations are seed commitments and seed expansions (via the PRG), both of which require the computation of $2 \cdot \tau \cdot N$ AES-128 blocks, where N denotes the number of leaves per tree and τ the number of parallel repetitions. The tree expansion is approximately twice as cheap, as it requires computing only about $\tau \cdot N$ AES-128 blocks.

For the trade-off "Faster" ($N := 64$), the costs of the BLC and PIOP computations become comparable. Further reducing N would make the PIOP the dominant computational bottleneck, since its cost increases (while the BLC cost decreases) as N decreases. Achieving significantly shorter running times with MQOM would therefore require further optimization of the matrix multiplications and the associated arithmetic operations.

Comparison with Other PQ Schemes Embedded Implementations. Table 5 summarizes the performance of representative existing implementations, used as a baseline for comparison.

We first compare our results with existing implementations of MPCitH-based schemes. Compared to the reference implementation of MQOM, our code achieves significantly faster execution times together with a much smaller memory footprint. In particular, for the reference MQOM code, only the signing procedure for the "L1 fast" parameter set can be executed on the Nucleo L4R5ZI platform, due to its strict memory requirements.

[11] Using more memory aggressive strategies with a basic DFS walk within a tree for the BLC, our implementation also achieves less than 6 KB for the footprint and less than 10 KB including the signature and key buffers. This nonetheless comes at the cost of almost doubling the cycles for the Short variant.

Table 5: Comparative performance of PQ schemes on Cortex-M4 embedded platforms (for L1 security level). The provided memory footprint only reports the internal workspace requirement (excluding signature and keys).

Scheme	Instance	Sig. size	Implem	Running times (in Mc)			Memory (in KB)		
				KGen	Sign	Verify	KGen	Sign	Verify
MPCitH based PQ candidates with embedded optimized implementations									
MQOM ($\mathbb{F}_{16}$ R5) **This work**	Faster	3 984	LUT	6.74*	29.2*	28.0	3.52*	12.0*	12.9
			Bal.	9.95	37.3	28.5	2.5	10.9	12.0
			Mem.	9.93	79.6	68.4	1.48	6.56	3.91
			Hard.	8.96†	23.5†	22.6†	1.28†	9.96†	10.9†
	Fast	3 280	LUT	6.74*	52.8*	46.4	3.52*	10.5*	11.8
			Bal.	9.95	73.2	47.0	2.5	9.50	10.7
			Mem.	9.93	102	70.9	1.48	5.76	3.91
			Hard.	8.96†	40.7†	32.2†	1.28†	8.58†	9.83†
	Short	2 916	LUT	5.99*	196*	194	3.65*	14.2*	17.7
			Bal.	9.23	291	194	2.65	13.5	16.8
			Mem.	9.21	366	212	1.62	6.00	4.24
			Hard.	7.94†	112†	108†	1.43†	12.2†	15.6†
MQOM Ref. ($\mathbb{F}_{16}$ R5)	Fast	3 280	Ref.	10.8	150	67.7	120	422	179
	Short	2 916	Ref.	10.5	-‡	255	142	1 700‡	599
FAEST-EM v1	Fast	5 696	[ADE+25]	0.08*	158*	154	0.81	31.3	31.1
FAEST-EM v1	Short	4 566	[ADE+25]	0.08*	1 288*	1 256	0.81	31.3	31.2
PERK v1-v2.0	Fast(5)	8 030	m4$^+$	0.82	237	78.1	9.03	25.2	21.8
PERK v1-v2.0	Short(5)	5 780	m4$^+$	0.82	1 250	428	9.03	28.6	26.1
NIST standardized (or soon to be) PQ schemes									
ML-DSA-44	-	2 420	m4fstack$^+$	1.80	12.1	3.24	4.41	5.08	2.71
			m4f$^+$	1.43	3.94	1.42	38.3	44.8	8.91
Falcon-512	-	666	m4f$^+$	67.7	22.5	0.40	14.3	42.0	2.98
SPHINCS$^+$-shake	Fast	17 088	clean$^+$	50.5	1 182	70.5	2.76	2.37	2.57
	Small	7 856	clean$^+$	3 231	24 554	24.4	2.99	2.46	1.87
Non-MPCitH on-ramp NIST round 2 candidates with embedded optimized implementations									
MAYO-1	-	454	m4f$^+$	9.57	16.9	11.0	179	251	253
UOV-Ip	pkc-skc	128	m4fstack$^+$	175	89.2	12	383	243	6.55
CROSS-RSDPG	Bal.	9 120	ultra$^\diamond$	0.20	31.8	13.7	1.54	26.4	18.8
			light$^\diamond$	0.20	25.9	13.7	1.54	68.0	19.0
			Size Opt$^\perp$	x	74.6	18.2	x	8.10	8.00
HAWK-512	-	555	mem$^\triangle$	52.3	2.80	1.41	7.68	2.56	3.58
			fast$^\triangle$	52.3	1.16	1.23	7.68	3.15	4.92

* The performance results rely on the LUT use over secret entries.

† The performance results rely on an AES-128 hardware accelerator, benchmarked on STM32F437 custom board.

‡ Reference implementation of MQOM – with all BLC trees and all PIOP matrices in memory.
Memory footprint for signing is a lower bound as it exceeds the 640 KB SRAM of the Nucleo L4R5ZI board.

$^+$The results are extracted from [KPR+].

$^\diamond$ Results from [HMK+24].

$^\perp$Results from [SGB+25]. Keygen performance not covered.
At the time of writing, a PR for [KPR+] is pending here: https://github.com/mupq/pqm4/pull/397.
We consider it as the same as [SGB+25] since the submitter of the PR is among the authors.

$^\triangle$Results from [BBD+24].

A broader comparison can be made with Round-1 MPCitH-based schemes whose reference implementations were ported to the [KPR+] framework in [KKPY24]. However, similarly to the MQOM reference code, only a limited subset of parameter sets can be executed on embedded platforms because of their high memory consumption.

The first work specifically targeting an optimized implementation of a NIST MPCitH-based candidate for embedded devices is [BBB+24d], which focuses on PERK (version 1, corresponding also to version 2.0). Since the performance profile of MPCitH schemes

Table 6: Performance of our MQOM optimized implementations against the reference implementation on Cortex-M4 for security level 5.

Scheme	Instance	Sig. Size	Implem	Running times (in Mc)			Memory (in KB)		
				KGen	Sign	Verify	KGen	Sign	Verify
MQOM ($\mathbb{F}_{16}$ R5) **This work**	Faster	15 864	LUT	95.8⋆	371⋆	364	7.7⋆	43.2⋆	46.7
			Bal.	124	458	366	6.68	42.2	45.8
			Mem.	125	1 347	1 497	2.33	18.9	7.99
	Fast	13 772	LUT	95.8⋆	466⋆	454	7.71⋆	31.5⋆	32.3
			Bal.	124	710	459	6.68	30.4	31.2
			Mem.	125	1 336	1 243	2.33	15.8	7.75
	Short	12 014	LUT	90.5⋆	1 660⋆	1 659	7.96⋆	36.0⋆	41.3
			Bal.	118	2 828	1 656	6.95	35.1	40.7
			Mem.	118	3 806	2 419	2.60	16.4	8.26
MQOM Ref. ($\mathbb{F}_{16}$ R5)	Fast	13 772	Ref.	-‡	-‡	-‡	780‡	1 180‡	827‡
	Short	12 014	Ref.	-‡	-‡	-‡	780‡	6 550‡	1 150‡

⋆ The performance results rely on the LUT use over secret entries.

‡ Reference implementation of MQOM – with all BLC trees and all PIOP matrices in memory. Memory footprints are lower bounds as they exceed the 640 KB SRAM of the Nucleo L4R5ZI board. Note that the PIOP matrices take twice the size given in Table 3 because their triangular shape is not exploited on `x86` where memory is not an issue.

inherently reflects a trade-off between signature size and running time, a fair comparison should be conducted for instances with the same number N of tree leaves. We therefore compare the "Short" instance of PERK (v2.0) with the "Fast" instance of MQOM, as both use $N := 256$. Under this comparison, our implementation clearly outperforms that of [BBB+24d], achieving roughly a tenfold improvement in signing time while reducing the memory footprint by a factor of at least 2-3.

The second work optimizing a NIST candidate for embedded devices is [ADE+25], which targets the Round-1 version of FAEST. Since the implementation in [ADE+25] relies on look-up tables with secret-dependent accesses, we compare it against our LUT-based profile. For the "Fast" instance, our implementation achieves approximately a threefold improvement in both signing and verification times, as well as in memory footprint.

We now compare our implementation with schemes that do not rely on the MPCitH paradigm. Unsurprisingly, lattice-based schemes (*e.g.*, ML-DSA, Falcon, and HAWK) are more efficient in terms of raw cycles performance. We can however observe that our code seems more memory-friendly than Falcon's optimized code, which has relatively large memory footprint (∼40 KB) for signing. Also, schemes such as ML-DSA that use Fiat-Shamir with aborts suffer from rejection sampling hazards, leading to a complex integration into hard real-time embedded systems (reasonable *guaranteed response time* for signing is difficult to evaluate). Compared to SPHINCS+, our implementation is significantly faster. Although SPHINCS+ has a low memory footprint, its performance is hindered by the large signatures it produces, which must also be stored in memory (in a setting without streaming).

UOV-like schemes typically exhibit smaller running times than MPCitH-based schemes, but their main limitation lies in their very large memory footprint, often reaching several hundreds of kilobytes. It is therefore unclear how these schemes would behave under stringent memory constraints. Finally, CROSS, a code-based scheme designed to achieve lighter computational performance than MPCitH-based schemes, indeed offers slightly faster running times. However, it appears to require a somewhat larger memory footprint, a drawback that becomes even more pronounced when accounting for the storage of the large signature itself.

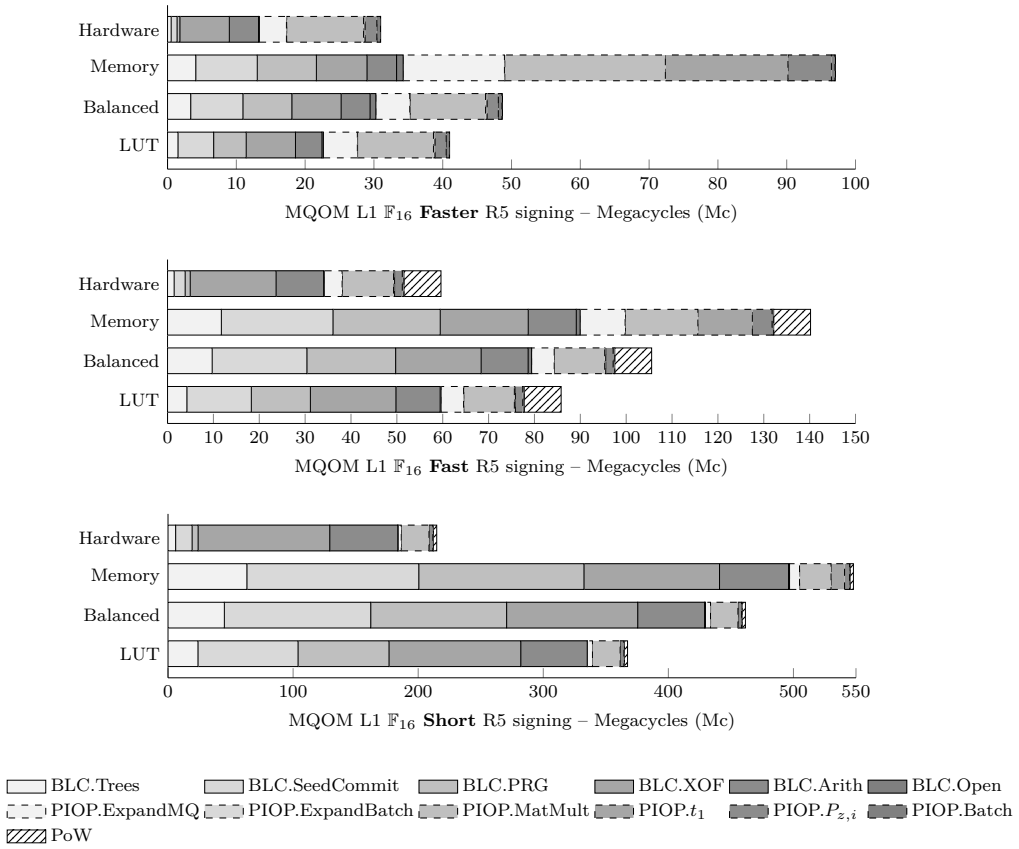


Figure 6: Detailed cycles performance of BLC and PIOP parts in the signature procedure for MQOM L1 $\mathbb{F}_{16}$ R5 faster, fast and short for each implementation variant LUT, Balanced, Mem and Hardware. Beware that the total number of cycles slightly differs from the results in Table 5 because of the additional cycles induced by all the measurement probes (the proportions of the parts of the algorithm remain the same though).

3.6 Further Discussions

The Sigma Variant. In Section 3, we focused on the R5 variant. All the proposed optimizations are applicable to the R3 variant. However, it is possible to go a bit further with R3. As there is no batching in this variant (equivalent to Γ being the identity matrix), com_1 is not used in ComputeAlphaLines. Therefore, it is possible to have a per-repetition computation for the signing algorithm in addition to the verification algorithm, as explained in the paragraph "Per-repetition computation" of Section 3.4.

Storing the Expanded Public Key in Flash. Expanding the public MQ equations tends to be a heavy operation. Therefore we should avoid expanding them too many times. As explained at the end of Section 3.3, we might want to expand them several times to reduce the SRAM usage, raising some dilemma. In some contexts, an interesting option could be to store the expanded MQ equations in Flash (either embedded or external to the MCU). It might be very interesting when the embedded device is supposed to generate/verify signatures for fixed keys, *e.g.* for local firmware or software updates authenticity verification. In this case, there is zero expansion cost in cycles and memory, and so the optimal trade-off memory-speed will be better than those proposed in Table 5.

Signature Size. In the benchmark of Tables 5 and 6, the memory footprint refers to the internal usage of the benchmarked algorithms. It does not take into account the sizes of the input/output buffers, *i.e.* the size of the keys and of the signature. While the (unexpanded) keys in MQOM are very small, it is not the case of the signature (3-4 KB for L1, 12-15 KB for L5). The current benchmarks do not use the input buffers as working space. More precisely, the signature buffer is used only to write the signature in the signing function, while it is used only to read the signature in the verification algorithm. While our code does not have a streaming API (except in the context of Section 5.1), we can thus approximate the memory requirement of a streamed implementation by the performance numbers of our benchmarks. In the case of a non-streamed implementation, we need to take into account that the signature will be stored in SRAM. However, if we allow using the signature buffer as a scratchpad memory (to store intermediary values), we can achieve even smaller memory footprints than those given in Tables 5 and 6. In our current implementation, we can achieve a signing internal memory size of 4.80 KB instead of 5.76 KB for the L1 Fast instance "Memory" implementation (at the same speed), for which we need to add the size of the signature and the secret key $3280 + 88$ bytes.

Recomputation. Our current optimized implementations do not rely much on recomputations. The recomputed operations are limited to key schedules and sometimes to the expansion of the MQ equations. More aggressive recomputation strategies are possible. At the cost of expensive recomputation, we might succeed to reduce even further the memory footprint. For example, [ADE+25] proposes a strategy to avoid storing at the same time all the polynomials in P_x and P_u for a single repetition. However, the price to pay is very high as it requires computing several times the tree expansion, the leaf expansion, etc. We believe that the memory footprint we succeeded to achieve with limited recomputation is already sufficient to most of the existing usage contexts.

Application to VOLE-in-the-Head. MQOM is based on the TCitH framework [FR25]. While Mirath and RYDE rely on the same framework, FAEST, PERK, and SDitH are built on the VOLEitH framework [BBD+23b]. Although these two frameworks are structurally very close, it is important to stress that VOLEitH-based schemes tend to require more memory than TCitH-based ones. The reason is that VOLEitH's PIOP does not rely on parallel repetitions, which prevents per-repetition computation and thus limits opportunities for memory reuse. Moreover, VOLEitH-based schemes make more intensive use of leaf expansion, since they must expand the λ-bit mask used to protect the VOLEitH consistency check. As a result, they also tend to be slightly slower in practice.

4 Using One-Tree Technique

While the batched line commitment of MQOM relies on correlated trees [HJ24, KLS25] to shorten signatures, the other NIST MPCitH-based candidates (FAEST, Mirath, PERK, RYDE, SDitH) use the one-tree technique [BBM+24]. Instead of having τ independent trees of N leaves, they build a single GGM tree with $\tau \cdot N$ leaves in which the leaves corresponding to the τ parallel repetitions are interleaved. Concretely, the i-th seed of repetition e is placed at leaf $(e \cdot N + i)$ of the large tree. One then reveals all leaves except τ, which is more efficient than revealing all but one leaf in τ independent trees: the hidden paths overlap, reducing the number of internal nodes that must be revealed.

4.1 Existing One-Tree Implementations

As in Section 3.2, having a memory-friendly code of the large GGM tree requires to implement an on-the-fly tree computation. However, providing an on-the-fly implementation of

such a tree is substantially more difficult compared to the trees involved in Section 3.2, as we need to keep hidden several leaves (instead of just one). Moreover, as the leaves for all the repetitions are interleaved, it requires to perform the operations of the τ parallel repetitions at the same time, roughly multiplying by τ the memory footprint of BLC.Commit. For example, we will need τ contexts of incremental hashing to hash the seed commitments of the parallel repetitions. Such a context takes at least 200 bytes in memory (the size of a Keccak state), implying that the τ contexts cost already at least 3.4 KB for the instance L1 fast. Avoiding this factor τ would require to re-expand the tree many times, leading to a very high computational overhead. We do not investigate this option in this work, as we aim to keep decent running times. A mixed strategy would consist in computing per group of `nb_para_reps` repetitions.

Unfortunately, existing MPCitH candidates using the one-tree technique do not design their trees with on-the-fly computation in mind. The choice of the leaf ordering and the node ordering in the opening bitstring has important impacts on the on-the-fly implementation. The only known MPCitH implementation that supports an on-the-fly one-tree evaluation is the reference SDitH v2 implementation [ABB+24]. Because the SDitH tree structure is not stream-friendly, the resulting code is relatively complex and relies on some recomputation, while it still achieves a memory footprint of only a few hundred kilobytes – significantly smaller than that required to store the full tree in memory, but still too expensive for embedded devices. Indeed, since the order in which the revealed nodes are processed is not stream-friendly, the implementation must rely on heavy auxiliary data structures to reorganize nodes appropriately. For example, in SDitH v2 the partial-expansion routine stores an information structure for every node (`ggm_multi_item_t`), amounting to 4 bytes per node. Although this is far smaller than storing the fully expanded tree, it remains costly, $\approx$ 192 KB for the L1-short parameter set under MQOM parameters ($\approx$ 180 KB under SDitH parameters). Even storing just one bit per node (to track whether a node is revealed) becomes expensive: 3.0 KB for L1-short and 6.4 KB for L5-short. Let us stress that it is not clear that storing a single bit per node is enough to run the algorithm, but it provides a clear lower bound. To achieve smaller footprints, we need to have a memory usage that does not scale *linearly* with the number of nodes.

4.2 Designing an Embedded-Friendly One-Tree Technique

In this section, we investigate how to design a simple embedded-friendly GGM tree that hides multiple leaves efficiently. Three design choices must be made: (1) tree format, (2) leaf ordering, and (3) ordering of revealed nodes during opening. Our goal is to obtain a simple depth-first search/walk (DFS) implementation with minimal memory footprint.

Tree Format and Leaf Ordering. Since $\tau \cdot N$ is generally not a power of two (because τ is not), the GGM tree is not perfect. Several tree formats are possible. We focus on *full* trees – each internal node has exactly two children – to minimize the number of node derivations. CROSS [BBB+24a], which is not a MPCitH-based scheme but also requires non-perfect trees, constructs a forest of perfect trees because of its convenient recursive structure. All MPCitH candidates using the one-tree technique instead use full complete trees.

We adopt the full complete tree format as well, since it naturally fits an array representation: the children of node i are $2i$ and $2i + 1$, with the root at index 1. In such trees, leaves appear at two different depths. Two natural leaf orderings exist: by node index, or left-to-right order. For classical implementations that materialize all leaves in memory, the choice is largely irrelevant. For on-the-fly traversal, however, it is crucial. A left-to-right ordering enables a clean DFS traversal starting from the first leaf, whereas ordering by node index would require starting DFS from a non-extreme leaf, causing either

recomputation or additional memory because the paths to the first and last leaves are not disjoint.

All existing MPCitH schemes using the one-tree technique adopt the node-index ordering, which is natural when on-the-fly computation is not considered. In this work, we adopt the left-to-right ordering.

Routines and Path Ordering. Once the tree format and leaf ordering are fixed, the full expansion routine (expanding all leaves from the root) is straightforward using DFS. It requires only storing the current root-to-leaf path of length at most $h+1$ where h is the tree height ($h := \lceil \log_2(\tau \cdot N) \rceil$), *i.e.* storing $(h+1)\lambda$ bits.

The more challenging tasks are producing the opening and performing partial expansion. Their efficiency depends crucially on the order in which revealed nodes are emitted. In all existing MPCitH candidates that employ the one-tree technique, the revealed nodes are ordered by tree depth. However, this ordering is not stream-friendly: it does not match the order in which a depth-first search (DFS) naturally encounters nodes. As a consequence, implementations must rely on heavy auxiliary data structures to reorder nodes appropriately (see Section 4.1 for the SDitH example). Our goal is to have a memory footprint that does not scale linearly with the number of nodes, without incurring extra computational overhead. For this reason, our design adopts a left-to-right ordering of the revealed nodes – the same order in which a DFS traverses the tree. This removes the need for large auxiliary data structures and allows stream-friendly processing. The remaining challenge is to determine how to efficiently compute the set of revealed nodes in this order.

Our solution introduces a routine `get_hidden_nodes` that:

- takes as input the ordered list of the τ hidden leaves, and
- outputs a list L of all node indices that must remain hidden, sorted exactly in the order in which a left-to-right DFS would encounter them.

This routine is easy to implement: for each hidden leaf, traverse its path from root to leaf and add all nodes on that path. Redundant nodes are removed by comparing with the previous path (requiring storing only h node indices). The memory required is at most $\tau \cdot (h+1)$ node indices in L, and a table of h indices storing the previous path. Using 2 bytes per node index, the total cost is: ≤ 414 bytes for L1-short, and ≤ 882 bytes for L5-short. Using this list L, both the opening computation and the partial expansion can be implemented as a clean DFS that simply outputs all the first nodes not in L (or expands them) as it traverses the tree. Only L and the current root-to-leaf path of $h+1$ λ-bit seeds must be stored. The total footprint is of $\leq 2 \cdot \tau \cdot (h+1) + h \cdot (\lambda/8)$ bytes, *i.e.* 0.6-0.7 KB for L1 and 1.4-1.5 KB for L5.

Opening Threshold and Padding. An important observation when hiding multiple leaves in a GGM tree is that the number of revealed nodes varies, which naturally leads to variable signature sizes. To mitigate this, [BBM+24] suggests fixing a threshold parameter T_{open} and sampling a new opening challenge whenever the number of revealed nodes exceeds T_{open}. However, in many cases the number of revealed nodes is smaller than T_{open}.

To enforce a constant signature size, FAEST, Mirath, PERK, RYDE, and SDitH adopt *padding*: they append zero bits so that the bitstring encoding the revealed nodes always has length $T_{\text{open}} \cdot \lambda$. This approach, however, has a major drawback: the verifier must check that all padding bits are indeed zero. Otherwise, the scheme fails to satisfy strong existential unforgeability under chosen-message attacks (sEUF-CMA), since an adversary could modify the padding of a valid signature to produce a distinct valid signature for the same message. FAEST and SDitH explicitly require the verifier to validate the padding. Mirath, PERK, and RYDE do not specify padding in their documentation, but their official implementations use it and include the padding check. Even when padding is formally

specified, implementers may easily forget this subtle but essential check, leading to security degradation.

In our work, we aim to avoid padding which is prone to incorrect implementations, while keeping constant signature size. Since the signer has a fixed budget of T_{open} nodes to reveal, our approach is simply to always use the full budget. Even when the minimal number of revealed nodes is less than T_{open}, nothing prevents us from intentionally revealing additional nodes. Concretely, our scheme slightly adjusts the termination condition of the tree traversal so that exactly T_{open} nodes are gathered. The subroutine `get_hidden_nodes` computes (at no extra cost) the minimal number of revealed nodes t. We then parameterize the depth-first search in the opening (and in the partial expansion) to explore exactly $T_{\text{open}} - t$ extra nodes. This approach eliminates the need for padding, avoids the associated security pitfalls, and reduces verifier cost, since every additionally revealed node saves the verifier one node-derivation operation.

Conclusion. In summary, we have shown how to instantiate the one-tree technique using a memory-efficient, embedded-friendly, padding-free approach. This avoids the subtle yet severe padding pitfalls present in some existing schemes, while keeping both signing and verification efficient.

4.3 Implementing the One-Tree Technique

We implemented a tweaked version of MQOM using the one-tree technique. As the MQOM candidate does not specify how to implement the one-tree technique, we choose to rely on the memory-friendly design proposed in the previous section. To have a fair comparison between the one-tree technique and the usage of correlated trees, we also rely on the half-tree technique [GYW+23, CLY+24, BCD24b]. Since one needs to keep hidden several root-to-leaf paths, the domain separation for node derivation should be based on the node index rather than the repetition/depth pair used in correlated trees. The one-tree technique thus prevents us to factorize some key schedule computations between nodes by design. Moreover, let us highlight that the cost of the pseudo-random expansion of the leaf is more expensive as we need to generate a mask of the λ first bits of the witness, while it is not the case when using correlated trees. Regarding the tree traversal strategy, we rely on the advanced walk described in Section 3.2 (with $p = 2$) for signing, and on a basic DFS traversal for verifying. Parallel walks are not applicable in this setting, as there is only a single tree, and in-place subtree computation provides limited benefits, since most of the key schedules cannot be factorized.

We report performance results in Table 7 for the trade-offs "Fast" and "Short". We use the same implementation profiles than in Section 3.5. As expected, one achieves larger memory footprints (due to the computation of several repetitions at the same time) and slower routines (due to the non-factorized key schedules of the tree and more expensive leaf expansions). We can observe that increasing the number of tree computations $\lceil \frac{\tau}{\texttt{nb_para_reps}} \rceil \geq 1$ indeed allows us to achieve smaller memory footprint, at the cost of more cycles. To the best of our knowledge, our code is the first memory-friendly implementation of the one-tree technique, additionally bringing flexible performance trade-offs.

5 Exploring Alternative Performance Trade-Offs

In Section 3.5, we achieve very competitive running times with low memory footprint. The obtained performance outperforms all the previous existing implementations of MPCitH/VOLEitH-based schemes. It also offers interesting performance profiles compared to other post-quantum schemes. Unfortunately, it is still quite heavy and has relatively large signatures, especially when compared to pre-quantum schemes. In Section 5.1, we

Table 7: Performance of the tweaked MQOM scheme using the one-tree technique. The number of the implementation tag refers to the number of tree computations $\lceil \frac{\tau}{\texttt{nb_para_reps}} \rceil$.

Scheme	Instance	Sig. Size	Implem	Running times (in Mc)			Memory (in KB)		
				KGen	Sign	Verify	KGen	Sign	Verify
MQOM (L1 $\mathbb{F}_{16}$ R5) **Using One-Tree**	Fast	3 264	LUT-1	6.75*	62.7*	56.3	3.52*	35.7*	36.0
			Bal.-1	9.95	88.2	56.4	2.50	34.4	35.0
			Mem.-1	9.93	186	109	1.48	11.7	11.2
			Hard.-1	8.96†	50.0†	43.6†	1.28†	18.2†	18.6†
			LUT-2	6.74*	68.4*	63.0	3.52*	22.4*	22.6
			Bal.-2	9.95	101	63.1	2.50	21.5	21.6
			Mem.-2	9.93	203	116	1.48	8.80	8.30
			Hard.-2	8.96†	55.1†	49.7†	1.28†	11.7†	12.1†
	Short	2 884	LUT-1	5.99*	246*	249	3.65*	28.6*	29.2
			Bal.-1	9.23	381	249	2.64	27.5	28.2
			Mem.-1	9.20	991	591	1.62	10.1	9.47
			Hard.-1	7.93†	179†	177†	1.42†	14.6†	15.5†
			LUT-2	5.99*	278*	288	3.65*	17.9*	21.3
			Bal.-2	9.23	459	457	2.64	17.0	20.4
			Mem.-2	9.20	1 070	636	1.62	7.96	7.24
			Hard.-2	7.93†	206†	210†	1.43†	12.8†	16.2†

* The performance results rely on the LUT use over secret entries.
† The performance results rely on an AES-128 hardware accelerator, benchmarked on STM32F437 custom board.

investigate how we can mitigate the signature size and the running time for the verifier. Then in Section 5.2, we investigate how pre-signatures (or offline/online decomposition) can mitigate the slow signing time.

5.1 Streaming the signature

Let us explore how we could mitigate the verification cost and the impact of the signature size. To deal with large messages, some signature implementations allow to stream the message. Instead of directly taking the entire message as input, they have an update routine that can process consecutive parts of the message. It is for example implemented in the `libsodium` library [Dc13] (*c.f.* "multi-part messages" paragraph in "Public-key signatures" page). We can imagine a similar process for the signature itself. It does not exist in the context of pre-quantum signatures for two reasons: (1) the verification is roughly a single operation (*e.g.* an exponentiation), (2) the signature size is short which limits the interest of the approach. However, the signature sizes of post-quantum schemes are much larger (up to some exceptions like SQIsign [AAA+24]) and the algorithms are much more complex. Streamed verification has already been studied for some post-quantum signature schemes, where the signature and the public key are sent in data chunks to a device with limited memory resources. In [GHK+21], the authors show how the signature verification of hash-based (SPHINCS$^+$), lattice-based (Dilithium) and MQ-based (Rainbow and GeMSS) schemes can fit in only 8 KB of SRAM and Flash on a Cortex-M3 through a streaming API, while keeping decent performance at a rate of 500 kbit/s. Specifically for SPHINCS$^+$, the authors of [NRW21] further optimize the size of the streamed chunks in a prototype implementation of a TPM, while also adding a streamed signature generation to the verification. In the MPCitH-based schemes, we often deal with parallel repetitions of some underlying identification scheme, and so we naturally have a (partial) independence between some internal blocks of the signature transcript.

Let us apply this idea of streamed verification to MQOM. Ideally, we would like to decompose the routines BLC.Eval and RecomputeAlphaLines respectively as described in Algorithm 7 and Algorithm 8, where BLC.Eval_x1 and RecomputeAlphaLines_x1 are the natural splitting per repetition. Unfortunately, the scheme as described in the official

specification does not ease this streaming, because the fragments of the signature transcript are not in a convenient order.

Algorithm 7
BLC.Eval(salt, opening, r)

1: $(\texttt{opening}_0 \parallel \ldots \parallel \texttt{opening}_{\tau-1}) := \texttt{opening}$
2: **for** $e = 0$ **to** $\tau - 1$ **do**
3: $(\texttt{hash_ls_com}[e], x_{\texttt{eval}}[e], u_{\texttt{eval}}[e])$ $\leftarrow$ BLC.Eval_x1(salt, $\texttt{opening}_e, r[e]$)
4: $\mathsf{com}_1 \leftarrow \mathsf{Hash}_7(\texttt{hash_ls_com})$
5: **return** $(\mathsf{com}_1, x_{\texttt{eval}}, u_{\texttt{eval}})$

Algorithm 8
RecomputeAlphaLines($\mathsf{com}_1, \alpha_1, r, x_{\mathsf{eval}}, u_{\mathsf{eval}}, \rho_{\mathsf{eq}}, \hat{y}$)

1: $\texttt{piop_data}[0] \leftarrow \mathsf{ExpandMQ}(\rho_{\mathsf{eq}}, \hat{y})$
2: $\texttt{piop_data}[1] \leftarrow \mathsf{ExpandBatching}(\mathsf{com}_1)$
3: **for** $e = 0$ **to** $\tau - 1$ **do**
4: $\alpha_0[e] \leftarrow$ RecomputeAlphaLines_x1($\texttt{piop_data}, \alpha_1[e]$,
5: $r[e], x_{\texttt{eval}}[e], u_{\texttt{eval}}[e]$)
return α_0

Figure 7: Split variants of BLC.Eval and RecomputeAlphaLines for verification streaming

Algorithm 9 Verif(pk, msg, sig)

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

$(\rho_{\mathsf{eq}}, \hat{y}) \leftarrow \texttt{pk}$
$\texttt{piop_data}[0] \leftarrow \mathsf{ExpandMQ}(\rho_{\mathsf{eq}}, \hat{y})$
$\texttt{chunk}_0 \leftarrow \texttt{ReadFromSig}(\texttt{sig}, 0)$
▷ First chunk
$(\texttt{hash}, \mathsf{salt}, \mathsf{com}_1, \texttt{nonce}) \leftarrow \texttt{chunk}_0$
$\texttt{piop_data}[1] \leftarrow \mathsf{ExpandBatching}(\mathsf{com}_1)$
$(r, \texttt{val}) \leftarrow \mathsf{DeriveChallenge}(\texttt{hash}, \texttt{nonce})$
▷ Following chunks
for $e = 0$ **to** $\tau - 1$ **do**
$(\texttt{opening}_e, \alpha_1[e]) \leftarrow \texttt{chunk}_{e+1} \leftarrow \texttt{ReadFromSig}(\texttt{sig}, e+1)$
$(\texttt{hash_ls_com}[e], x_{\texttt{eval}}[e], u_{\texttt{eval}}[e]) \leftarrow \mathsf{BLC.Eval_x1}(\mathsf{salt}, \texttt{opening}_e, r[e])$
$\alpha_0[e] \leftarrow \mathsf{RecomputeAlphaLines_x1}(\texttt{piop_data}, \alpha_1[e], r[e], x_{\texttt{eval}}[e], u_{\texttt{eval}}[e])$
▷ Finalizing
$\mathsf{com}_1' \leftarrow \mathsf{Hash}_7(\texttt{hash_ls_com})$
$\mathsf{com}_2 \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
$\texttt{hash}' \leftarrow \mathsf{Hash}_4(\texttt{pk}, \mathsf{com}_1, \mathsf{com}_2, \mathsf{Hash}_2(\texttt{msg}))$
if $(\mathsf{com}_1' \neq \mathsf{com}_1$ **and** $\texttt{hash}' \neq \texttt{hash}$ **and** $\texttt{val} \neq 0)$ **then return** False
return True

Figure 8: MQOM v2 – Streamable verification (ReadFromSig reads chunks from sig).

The ideal order for elements contained in the signature transcript is the following. All the elements that are *common* to all parallel repetitions should be at the beginning of the signature. This includes the salt, the hash digests of the Fiat-Shamir challenges and the proof-of-work nonce. In MQOM v2, the nonce is at the end of the signature, preventing us to derive the opening challenge r without getting the entire signature. Those common elements will constitute the first chunk $\texttt{chunk}_0$ of the signature. Then, the following chunks $\texttt{chunk}_1, \ldots, \texttt{chunk}_\tau$ should contain the elements *specific* to a parallel repetition. From $\texttt{chunk}_0$ and $\texttt{chunk}_e$, we should be able to check/compute the e^{th} parallel repetition. Those chunks correspond to the BLC opening (opening) and the PIOP output (α_1). In MQOM v2, the BLC openings of all the repetitions are interleaved, preventing us to stream data (a BLC opening contains three types of data and, in MQOM v2, these quantities are gathered per type, and not per repetition).

Let us remark that MQOM puts in the signature the hash digest com_1 from which the batching challenge is derived. While this quantity is somewhat redundant as the verifier recomputes it in BLC.Eval, it unlocks the possibility to stream the PIOP verification per repetition. Without it, it would be impossible to derive the batching challenge (in verif_data) without having the entire signature (in the case of the variant R3 of MQOM,

we do not care since there is no batching challenge). Finally, by replacing com_2 by $\mathtt{hash}$ in the signature, we can observe that we do not need anymore the message to start the verification. We describe in Algorithm 9 the streamable verification.

Remark 4. We explained above that we need to change the order of the elements in the MQOM signature to be streamable. In practice, we need to do an additional tweak of the specifications. In the original version of MQOM, the input of Hash_7 in $\mathsf{BLC.Eval}$ includes some auxiliary values $\Delta_x^{(1)}[e]$ (one per repetition), but to have the API described in Algorithm 7, it would only require to move the hashing of those values in a hash operation that is inside $\mathsf{BLC.Eval_x1}$ (inside Hash_6 to be precise).

We implemented this tweaked/streamable version of MQOM. Let us stress that the mentioned changes do not impact the unforgeability security of MQOM. We give in Table 8 some benchmarks of the streamed verification. Let us observe that we can check each chunk $\mathtt{chunk}_e$ ($e > 0$) in any order, or in parallel. Therefore, the verification is highly parallelizable and we can send each chunk $\mathtt{chunk}_e$ in different communication packets (*e.g.* IP packets) out-of-order and start the verification before receiving all the chunks. Let us stress that the (2λ) first bits of the signature, denoted $\mathtt{hash}$, can be used as a signature identifier. Knowing two signatures with the same $\mathtt{hash}$ would imply knowing a SHAKE hash collision. We present on Table 8 the results for benchmarking the cycles of the LUT based MQOM implementation for L1 and L5: as we can observe, the global performance in cycles (sum of initialization, dealing with chunks and finalization) is mostly preserved when compared to the non-streaming verification. An interesting observation is that since dealing with each $\mathtt{chunk}_e$ can be performed in parallel, the cost of these chunks processing will correspond to $\approx \frac{\tau}{p} \times \mathtt{latency}_{\mathtt{chk}_e}$ where $\mathtt{latency}_{\mathtt{chk}_e}$ is the latency of processing each non-zero chunk and $0 < p < \tau$ is the parallelism level of the platform. This allows to reduce the verification global duration: for instance on the L5 Short instance, we can decrease this latency from 1706 Mc to $\approx$ 956 Mc[12] with $p = 2$, which can be a realistic scenario on embedded platforms where dual-MCU cores communicate in the same SoC over a shared AXI bus (*e.g.* the STM32H7 SoC family embeds a Cortex-M7 and a Cortex-M4, many RISC-V SoCs are multi-core, etc.). Beyond the embedded domain, this type of parallelism can be even more effective on desktop-class $\mathtt{x86}$ and ARM platforms.

Table 8: Benchmark of the streaming verification of MQOM ($\mathbb{F}_{16}$ R5), assuming that the expanded MQ matrices are expanded only once in the initialization phase. Since the verification is not concerned by side-channel attacks, we used LUT based T-table Rijndael and log-exp field arithmetic. The running times are given in megacycles Mc.

	τ	$\lvert\mathtt{chk}_0\rvert$	$\lvert\mathtt{chk}_e\rvert$	$\lvert\mathsf{sig}\rvert$	Init.	Update		Final.	Total
						$\lvert\mathtt{chk}_0\rvert$	$\lvert\mathtt{chk}_e\rvert$		
L1 Short	12	84 B	236 B	2 916 B	3.49	0.03	15.4	0.10	188.0
L1 Fast	17	84 B	188 B	3 280 B	3.72	0.04	2.76	0.12	48.0
L5 Short	25	164 B	474 B	12 014 B	68.9	0.09	68.2	0.33	1 706
L5 Fast	36	164 B	378 B	13 772 B	69.5	0.16	15.0	0.45	595

Remark 5. The above stream-friendly property comes from the fact that the underlying identification scheme is repeated τ times in parallel. The VOLEitH-based schemes are less stream-friendly as only the construction of the small VOLEs are repeated. Moreover, the one-tree technique highly reduces the possibility of streaming. Indeed, the largest part of the signature corresponds to the sibling paths in GGM trees. When considering τ independent trees, we have τ independent small sibling paths. When using one-tree, we merge those τ trees yielding a unique large sibling path which is not streamable.

[12]The overhead due to the parallel processing units synchronization in shared memory is omitted here. Also, this figure supposes the same latency for the parallel tasks, which can vary when the execution cores are heterogeneous.

Remark 6. In Table 8, we expanded the MQ matrices once in the initialization phase. This choice has an important impact on the memory footprint, as we need to store in memory the entire MQ matrices (45 KB for L1 and 394 KB for L5, *c.f.* Table 3). A more memory-friendly choice would be to compute those matrices on-the-fly as in Section 3.3, but it would require expanding them τ times. While it would increase the total verification time by 20% for the short instance of L1, it would increase it by 80% for the fast instance. An alternative option would be compute several chunks at the same time to decrease the number of recomputations for the expansion. Implementing such trade-offs is left as future work.

Remark 7. In this section, we discussed about streaming the verification algorithm. One might also consider streaming the signing algorithm, in order to start outputting the signature before the signing procedure completes. Unfortunately, it is not possible/useful because most components of the signature are only determined at the end of the computation. As a result, streaming the signing algorithm would be beneficial only to avoid storing the full signature in memory.

5.2 Pre-signature

In Section 5.1, we have adapted a MPCitH-based signature verification to streaming which unlocks new use cases. Let us now explore how we could mitigate the signing cost. An idea would be to use the offline/online approach, which consists in splitting the signing algorithms in two phases:

1. offline: a first phase independent of the message to sign and which can be slow.
2. online: a second phase that uses the precomputed data to quickly produce a signature for a given message.

More precisely, the signing process would involve two algorithms $\mathsf{PreSign}$ (performed offline) and $\mathsf{FinalizeSign}$ (performed online):

$$\begin{aligned} \mathtt{presig} &\leftarrow \mathsf{PreSign}(\mathtt{sk}) \\ \mathsf{sig} &\leftarrow \mathsf{FinalizeSign}(\mathtt{sk}, \mathtt{presig}, \mathtt{msg}) \ , \end{aligned}$$

where $\mathtt{sk}$ is the private key, $\mathtt{presig}$ the precomputed material, $\mathtt{msg}$ the message to sign and sig the produced signature. Using this offline/online approach, we can drastically reduce the latency between receiving a message and producing a signature. This approach has been proposed in [EGM90]. In this seminal work, the authors showed how to build an efficient offline-online signature scheme using an ordinary signature scheme and a one-time signature scheme. However, some signature schemes naturally support the offline-online splitting. This is the case of ECDSA, where the signer can precompute from the nonce k and the private key x the values k^{-1} and $\mathtt{x}(k \cdot G)$, without having the message. This precomputation process has been implemented in some standard libraries, like in the historical API of OpenSSL [ope] (*c.f.* `ECDSA_sign_setup`).

In the case of the MPCitH-based schemes, we can easily rely on the offline-online approach. If the message is used only to derive the opening challenge, then we have a very efficient online phase, since it would just consists of revealing some already-computed quantities. This observation is not new: [AGH+23] already stated it. However, the limiting factor that was preventing an efficient offline-online signature scheme was the *size of the precomputed data*, which was historically high. It is due to the fact that we needed to store the views of many emulated parties. The offline-online split described in [AGH+23] was implemented for the hypercube variant of the first-round submission of the NIST candidate SDitH [AFG+23]. We can observe that the precomputed data for L1 in this implementation was of 103 KB, but we could easily go down to 11 KB by optimizing more the implementation. Moreover, by slightly tweaking the scheme, we would be able to

go down to 6.1 kilobytes. Let us stress that [AGH+23] was already using some modern MPCitH techniques. With older techniques (*e.g.* ZKBoo [GMO16], KKW [KKW18], etc.), we would obtain larger precomputed data.

Since [AGH+23], the state of the art of the MPCitH paradigm has evolved thanks to new frameworks TCitH [FR23, FR25] and VOLEitH [BBD+23b], leading to shorter signatures. We can observe that it also improves the offline-online concept on the MPCitH paradigm by reducing the size of the precomputed material. We give those sizes in Table 9 (column "with native PoW") for all the round-2 NIST MPCitH-based candidates. We can observe that we can go as small as around 500 bytes for MQOM, so we roughly gain a factor 10 compared to older MPCitH frameworks. We give in Algorithms 10 and 11 the corresponding routines PreSign and FinalizeSign for MQOM. Let us mention that the size of the precomputed data is given assuming that the message is only used for the opening challenge. This is not the case for PERK and FAEST, requiring to rely on a tweaked version of those schemes (the sizes would be much worse for the original versions). Let us stress that all the given sizes for precomputed data assume that we only put the master seed in the precomputed material, allowing to recompute the GGM trees in FinalizeSign.

In 2024, [BBM+24] suggests to rely on grinding/proof of work (PoW) to reduce the signature sizes. This approach is used in all the round-2 candidates. Unfortunately, this decreases the efficiency of the online signing. While using a PoW shortens the signature, it burdens the routine SampleChallenge. Having a PoW of w bits implies that we need to make around 2^w hash operations in average inside SampleChallenge. We give in Table 9 the bit size of the used PoW (for the schemes using the one-tree technique, the given number corresponds to the combination of the rejection due to the tree and the explicit grinding phase). We can observe that the bottleneck of the online signing time is clearly due to the PoW. For this reason, we give in Table 9 the sizes we would obtain when relying on small PoW (of less than 3 bits). We give in Table 10 the computational performance we obtained for MQOM L1 short and fast using the native PoW and the small one (we have used our Balanced implementation parameters of the signature here). As we can observe, the online cost is indeed very small (1.3% for the short variant, 13.8% for the fast variant) relatively to the offline phase where the heavy processing is performed. The smaller PoW allows to push further the gain and keep the online phase under 1 Mc for the two variants (the impact on the offline phase being almost zero for the fast variant, but non-negligible for the short one due to the heavier BLC).

Remark 8. As discussed in Section 4, there are two techniques to optimize the communication costs induced by the GGM trees: the one-tree technique [BBM+24] and the correlated trees [HJ24, KLS25]. MQOM is the only scheme based on correlated trees among the round-2 NIST candidates. Correlated trees lead to smaller precomputed data for pre-signatures. This can be easily explained by the fact that the one-tree technique reduces the size of the BLC opening (`opening`), which is not part of the precomputed material. On the other hand for correlated trees, the size of the auxiliary values in BLC key (`key`) is reduced, and these are part of the pre-signature. For instance, the pre-signature sizes for the MQOM variant based on the one-tree technique implemented in Section 4.3 are 656 bytes and 876 bytes, respectively for the trade-offs "Short" and "Fast" (to be compared with smaller 464 bytes and 604 bytes respectively for the correlated trees).

Remark 9. A pre-signature is as sensitive as the secret key itself, since we can recover the latter from the former. Hence, it is advisable to mask long-term pre-signatures before storing them, *e.g.* in non-volatile memory. In Algorithm 10, Line 8, we propose to sample a random value `rnd`, which is then combined with the secret key `sk` at Line 9 as a seed for a PRG. The PRG expands this seed into a mask `mask` of appropriate length, which is used as a one-time pad to protect `data`. The value `rnd` is included in the pre-signature so that Algorithm 11 can recompute `mask` and recover `data`. To keep the collision probability negligible over up to 2^{64} signatures generated with the same secret key, we conservatively

choose a size of 32 bytes for rnd at all security levels.

Algorithm 10 PreSign(sk)

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

1: $(\mathsf{pk}, x) \leftarrow \mathsf{sk}$
2: $(\rho_{\mathsf{eq}}, \hat{y}) \leftarrow \mathsf{pk}$
3: $(\mathsf{mseed}, \mathsf{salt}) \leftarrow \$$
4: $(\mathsf{com}_1, \mathsf{key}, x_0, u_0, u_1) \leftarrow \mathsf{BLC.Commit}(\mathsf{mseed}, \mathsf{salt}, x)$
5: $(\alpha_0, \alpha_1) \leftarrow \mathsf{ComputeAlphaLines}(\mathsf{com}_1, x_0, u_0, u_1, x, \rho_{\mathsf{eq}})$
6: $\mathsf{com}_2 \leftarrow \mathsf{Hash}_3(\alpha_0, \alpha_1)$
7: $\mathsf{rnd} \leftarrow \$$
8: $\mathtt{data} \leftarrow (\mathsf{salt}, \mathsf{com}_1, \mathsf{com}_2, \alpha_1, \mathtt{key})$
9: $\mathtt{mask} \leftarrow \mathsf{PRG}(\mathsf{rnd} \parallel \mathtt{sk})$
10: $\mathtt{masked_data} \leftarrow \mathtt{data} \oplus \mathtt{mask}$
11: **return** $\mathtt{presig} := (\mathsf{rnd} \parallel \mathtt{masked_data})$

Algorithm 11 FinalizeSign(sk, presig, msg)

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

1: $(\mathsf{rnd} \parallel \mathtt{masked_data}) \leftarrow \mathtt{presig}$
2: $\mathtt{mask} \leftarrow \mathsf{PRG}(\mathsf{rnd} \parallel \mathtt{sk})$
3: $\mathtt{data} \leftarrow \mathtt{masked_data} \oplus \mathtt{mask}$
4: $(\mathsf{salt}, \mathsf{com}_1, \mathsf{com}_2, \alpha_1, \mathtt{key}) \leftarrow \mathtt{data}$
5: $\mathtt{hash} \leftarrow \mathsf{Hash}_4(\mathtt{pk}, \mathsf{com}_1, \mathsf{com}_2, \mathsf{Hash}_2(\mathtt{msg}))$
6: $(i^*, \mathtt{nonce}) \leftarrow \mathsf{SampleChallenge}(\mathtt{hash})$
7: $\mathtt{opening} \leftarrow \mathsf{BLC.Open}(\mathtt{key}, i^*)$
8: **return** $\mathtt{sig} :=$
9: $\quad (\mathsf{salt}, \mathsf{com}_1, \mathsf{com}_2, \alpha_1, \mathtt{opening}, \mathtt{nonce})$

Figure 9: Pre-signing Algorithms.

Table 9: Pre-signature data sizes for MPCitH schemes with original and smaller PoW.

Scheme - Trade-off	With Native PoW			With Small PoW		
	PoW	|pre-sgn|	|sgn|	PoW	|pre-sgn|	|sgn|
MQOM $\mathbb{F}_{16}$ - Short	8	464 B	2 916 B	0	492 B	3 152 B
MQOM $\mathbb{F}_{16}$ - Fast	9	604 B	3 280 B	2	632 B	3 468 B
Mirath $\mathbb{F}_2$ - Short	11.3	790 B	2 990 B	2.8	847 B	3 255 B
Mirath $\mathbb{F}_2$ - Fast	11.1	1 064 B	3 456 B	3.0	1 120 B	3 720 B
RYDE - Short	10.4	883 B	3 115 B	2.8	948 B	3 355 B
RYDE - Fast	11.0	1 205 B	3 597 B	3.0	1 269 B	3 869 B
PERK* - Short	12.3	1 465 B	3 473 B	2.8	1 588 B	3 756 B
PERK* - Fast	9.8	2 080 B	4 312 B	2.9	2 203 B	4 451 B
SDitH - Short	11.9	1 685 B	3 705 B	2.8	1 828 B	3 992 B
SDitH - Fast	8.6	2 400 B	4 484 B	2.7	2 400 B	4 660 B
FAEST-EM* - Short	11.9	1 966 B	3 906 B	2.8	2 136 B	4 300 B
FAEST-EM* - Fast	8.3	2 816 B	5 060 B	2.9	2 986 B	5 230 B

* Reported pre-signature sizes suppose a tweak of the original schemes.

Table 10: Pre-signature performance (in Megacycles) of the offline PreSign and online FinalizeSign algorithms for MQOM L1 ($\mathbb{F}_{16}$ R5) short and fast, with original PoW and smaller PoW. The Balanced implementation trade-off have been used for this benchmark.

	Sig. Size	Pre. Size	Offline	Online
MQOM $\mathbb{F}_{16}$ Short R5	2 916 B	464 B	288	3.84
MQOM $\mathbb{F}_{16}$ Short R5 - LowPoW	3 152 B	492 B	309	0.91
MQOM $\mathbb{F}_{16}$ Fast R5	3 328 B	604 B	66.4	6.91
MQOM $\mathbb{F}_{16}$ Fast R5 - LowPoW	3 516 B	632 B	69.3	0.98

Let us now review the offline/online capabilities of other (than MPCitH) standards and candidates of the NIST competition.

ML-DSA. Without taking the rejection sampling into account, a pre-computed data would include at least the vectors y and w. For ML-DSA-44 and ML-DSA-87, it takes respectively around 5 250 and 10 368 bytes. Moreover, because of the rejection sampling due to Fiat-Shamir-with-aborts, a single couple (y, w) does not ensure to produce a valid signature, and so in practice, we would need to consume several couples[13], increasing the

[13] ≈ 4.25 for ML-DSA-44 and ≈ 3.85 for ML-DSA-87 on average.

memory footprint of the pre-computation approach for ML-DSA. The situation would be better for lattice-based schemes based on the Fiat-Shamir transformation without aborts, but the memory footprint would remain quite large. For example, for Raccoon [dPKPR24] with one masking share, the pre-signature would include (r, w), yielding 12 096 bytes for L1 and 21 248 bytes for L5.

SPHINCS$^+$/SLH-DSA. Using pre-computation does not help to mitigate the high signing cost of a hash-based signature, because the message knowledge is required as of the very beginning of signing.

Hash-and-sign paradigm. The post-quantum signature schemes following the hash-and-sign paradigm (such that FN-DSA and UOV-like schemes) do not behave well with the offline/online model. While we can rely on pre-computation to speed-up the online signing, the memory footprint of the pre-computed data is very high. For example, for MAYO [BCC+24a] and UOV [BCD+24a], the signature mostly consists in matrix multiplications and solving systems of linear equations involving vectors that are generated from the message through hashing and expanding: no pre-computation can be performed here without the message. The best that could be achieved is the unpacking of the secret key in memory from its compact form, which takes 237 KB for UOV L1 ($\approx$ 2.5MB for L5), and 141 KB for MAYO L1 (822 KB for L5).

Other Fiat-Shamir-based schemes. Beyond lattice-based and MPCitH-based schemes, there exists many other signature schemes based on the Fiat-Shamir transformation. Among the round-2 candidates to the second NIST call, there are LESS [BBB+24b], CROSS [BBB+24a] and SQIsign [AAA+24]. Let us discuss about those three schemes:

- **CROSS.** Applying the offline/online to CROSS would require to keep at least all the leaves of the Merkle trees among the pre-computed data (recomputing them would imply a lot of recomputation). The size of the pre-computed data is upper bounded by $(2\lambda) \cdot t$ bits, which is of 4 704 bytes (when targeting fast instances) and of 16 384 bytes (when targeting small signatures).
- **LESS.** Regarding LESS, the optimal pre-computed data (optimal in terms of memory footprint, while having fast online phase) would probably contain the possible responses for $\mathsf{rsp}^{(i)}$. There are $2t$ possible responses, of size of n bits. Therefore, the memory footprint of the pre-computed data would be of 2 835 bytes for the more convenient L1 instance (LESS-252-45), and of 12 096 bytes for the less convenient L1 instance (LESS-252-192).
- **SQIsign.** The pre-computed data for SQIsign would contain the commitment ideal I_{com}, the commitment curve E_{com}, and the points $(P_{\text{com}}, Q_{\text{com}})$. The curve and the points can be represented with three $\mathbb{F}_{p^2}$ elements, which costs 186 bytes in total for L1. The ideal costs additional 160 bytes. In total, the memory footprint of the pre-computed data would be of 356 bytes for L1.

Without surprise, SQIsign is the most memory friendly option among the post-quantum schemes, as it is well-known to have very compact data. Otherwise, the MPCitH-based schemes are the second best option in term of memory footprint for the pre-computed data. MQOM achieves the smallest footprint among them with $\approx$ 500 bytes of pre-computation for L1, quite close to the SQIsign scheme. For larger security levels, the difference would be a bit larger, as the object sizes grow linearly in SQIsign while they grow quadratically for MPCitH-based schemes.

6 Conclusion

In this work, we proposed portable optimized implementations of the NIST candidate MQOM targeting embedded platforms with low memory constraints. We presented various optimizations of the BLC and of the PIOP parts of the algorithm, which represent the heavy processing during signature and verification. These optimizations allow for various trade-offs between very low memory consumption and efficiency in terms of cycles, giving the flexibility of achieving the best performance on different MCU families as well as different usage contexts.

Beyond MQOM, many of our optimization strategies apply to other MPCitH-based schemes. While MQOM relies on correlated trees, we also provided a memory-efficient implementation of the one-tree technique. Although it is possible to implement the tree with very low memory usage (assuming that on-the-fly computation is carefully integrated into the tree design), the one-tree technique suffers from a major drawback: multiple parallel repetitions must be handled simultaneously. As a result, it generally leads to slower algorithms and a larger memory footprint compared to correlated trees. Moreover, it is more complex to implement due to path merging and signature-size variability, while also preventing verification streaming and slightly increasing the size of pre-signatures.

This work highlights that MPCitH-based schemes offer several implementation advantages compared to other signature families. In particular, they can be memory-friendly, clearly outperforming UOV-like schemes in this respect. When the signature buffer must be stored in memory, their performance is competitive with CROSS. In terms of running times, they outperform SPHINCS$^+$, and one can reasonably expect them to outperform the NIST candidate LESS, which involves heavy computations –although no embedded-device implementation of the latest version of LESS is currently available. While MPCitH-based schemes may be competitive with Falcon in highly memory-constrained environments, they remain overall less efficient than lattice-based schemes. However, due to their structural design, they outperform almost all other signature families (including lattice-based one) in contexts where precomputation is possible.

Moreover, as mentioned in the introduction, MPCitH-based schemes also exhibit clear implementation advantages: they avoid floating-point arithmetic (cf. Falcon), Gaussian elimination (cf. UOV-like schemes and LESS), and rejection sampling (cf. ML-DSA). In addition, all NIST MPCitH-based candidates now rely exclusively on binary fields, rather than prime fields, thereby avoiding the need for arithmetic-Boolean conversions (cf. most lattice-based schemes, LESS, and CROSS). As a result, one may reasonably expect MPCitH-based schemes to behave well under masking. Future work will therefore focus on analyzing the side-channel leakage of our implementation and on designing protected versions of the code.

Acknowledgements. This work was funded by the French government through the France 2030 program (RESQUE project). Views and opinions expressed are however those of the authors only.

References

[AAA+24] 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. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[ABB+23] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, and Jean-Pierre Tillich. PERK. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[ABB+24] 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. SDitH — Syndrome Decoding in the Head. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[ABC+24] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, David Cooper, Quynh Dang, Thinh 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 first round of the additional digital signature schemes for the nist post-quantum cryptography standardization process. NIST Internal Report, NIST IR 8528, 2024.

[ADE+25] Diego F. Aranha, Johan Degn, Jonathan Eilath, Kent Nielsen, and Peter Scholl. FAEST for memory-constrained devices with side-channel protections. Cryptology ePrint Archive, Report 2025/1261, 2025.

[ADKF70] V. L. Arlazarov, Y. A. Dinitz, M. A. Kronrod, and I. A. Faradzhev. On economical construction of the transitive closure of an oriented graph, 1970.

[Ado23] Alexandre Adomnicai. An update on keccak performance on ARMv7-m. Cryptology ePrint Archive, Paper 2023/773, 2023.

[AFG+23] Carlos Aguilar-Melchor, Thibauld Feneuil, Nicolas Gama, Shay Gueron, James Howe, David Joseph, Antoine Joux, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. SDitH — Syndrome Decoding in the Head. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[AGH+23] Carlos Aguilar-Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The return of the SDitH. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 564–596. Springer, Cham, April 2023.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AP20] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, Dec. 2020.

[BBB+24a] Marco Baldi, Alessandro Barenghi, Michele Battagliola, Sebastian Bitzer, Marco Gianvecchio, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Edoardo Signorini, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger. CROSS — Codes and Restricted Objects Signature Scheme. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BBB+24b] Marco Baldi, Alessandro Barenghi, Luke Beckwith, Jean-François Biasse, Tung Chou, Andre Esser, Kris Gaj, Patrick Karl, Kamyar Mohajerani, Gerardo Pelosi, Edoardo Persichetti, Markku-Juhani O. Saarinen, Paolo Santini, Robert Wallace, and Floyd Zweydinger. LESS — Linear Equivalence Signature Scheme. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BBB+24c] Carsten Baum, Lennart Braun, Ward Beullens, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BBB+24d] Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Marco Palumbi, and Lucas Pandolfo Perin. Enabling PERK and other MPC-in-the-head signatures on resource-constrained devices. *IACR TCHES*, 2024(4):84–109, 2024.

[BBD+23a] Carsten Baum, 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. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

[BBD+23b] 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, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 581–615. Springer, Cham, August 2023.

[BBD+24] 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. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BBFR24] Ryad Benadjila, Charles Bouillaguet, Thibauld Feneuil, and Matthieu Rivain. MQOM — MQ on my Mind. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[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, *ASIACRYPT 2024,*

Part I, volume 15484 of *LNCS*, pages 463–493. Springer, Singapore, December 2024.

[BCC+24a] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BCC+24b] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized implementations for AVX2 and cortex-M4. *IACR TCHES*, 2024(2):252–275, 2024.

[BCD+24a] 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. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[BCD24b] Dung Bui, Kelong Cong, and Cyprien Delpech de Saint Guilhem. Improved all-but-one vector commitment with applications to post-quantum signatures. Cryptology ePrint Archive, Report 2024/097, 2024.

[BCH+23] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and vinegar: Modern parameters and implementations. Cryptology ePrint Archive, Paper 2023/059, 2023.

[CKY21] Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on cortex-m4. Cryptology ePrint Archive, Paper 2021/532, 2021.

[CLY+24] Hongrui Cui, Hanlin Liu, Di Yan, Kang Yang, Yu Yu, and Kaiyi Zhang. ReSolveD: Shorter signatures from regular syndrome decoding and VOLE-in-the-head. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part I*, volume 14601 of *LNCS*, pages 229–258. Springer, Cham, April 2024.

[Dc13] Frank Denis and contributors. The sodium cryptography library. https://doc.libsodium.org/, 2013. Accessed on October 2025.

[dPKPR24] Rafaël del Pino, Shuichi Katsumata, Thomas Prest, and Mélissa Rossi. Raccoon: A masking-friendly signature proven in the probing model. Cryptology ePrint Archive, Paper 2024/1291, 2024.

[EGM90] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital schemes. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 263–275. Springer, New York, August 1990.

[FR23] Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part I*, volume 14438 of *LNCS*, pages 441–473. Springer, Singapore, December 2023.

[FR25] Thibauld Feneuil and Matthieu Rivain. Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments. *Journal of Cryptology*, 38(3):28, July 2025.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Berlin, Heidelberg, August 1987.

[GHK+21] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying post-quantum signatures in 8 kB of RAM. Cryptology ePrint Archive, Paper 2021/662, 2021.

[GMO16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for Boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1069–1083. USENIX Association, August 2016.

[GYW+23] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in COT and DPF. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 330–362. Springer, Cham, April 2023.

[HJ24] Janik Huth and Antoine Joux. MPC in the head using the subfield bilinear collision problem. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part I*, volume 14920 of *LNCS*, pages 39–70. Springer, Cham, August 2024.

[HMK+24] Harry Hart, Puja Mondal, Suparna Kundu, Supriya Adhikary, Angshuman Karmakar, and Chaoyun Li. LightCROSS: A secure and memory optimized post-quantum digital signature CROSS. Cryptology ePrint Archive, Paper 2024/1929, 2024.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[KKPY24] Matthias J. Kannwischer, Markus Krausz, Richard Petri, and Shang-Yi Yang. pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Report 2024/112, 2024.

[KKW18] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 525–537. ACM Press, October 2018.

[KLS25] Seongkwang Kim, Byeonghak Lee, and Mincheol Son. Relaxed vector commitment for shorter signatures. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part IV*, volume 15604 of *LNCS*, pages 427–455. Springer, Cham, May 2025.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[KSSW22] Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

[NRW21] Ruben Niederhagen, Johannes Roth, and Julian Wälde. Streaming SPHINCS+ for embedded devices using the example of TPMs. Cryptology ePrint Archive, Paper 2021/1072, 2021.

[ope] OpenSSL `ECDSA_sign` and `ECDSA_sign_setup`. https://docs.openssl.org/3.3/man3/ECDSA_sign/#description.

[Por] Thomas Pornin. Bearssl's constant time aes implementation.

[SGB+25] Jonas Schupp, Marco Gianvecchio, Alessandro Barenghi, Patrick Karl, Gerardo Pelosi, and Georg Sigl. Optimizing the post quantum signature scheme CROSS for resource constrained devices. Cryptology ePrint Archive, Paper 2025/1928, 2025.

[SM16] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the internet and the open quantum safe project. Cryptology ePrint Archive, Paper 2016/1017, 2016.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and m4. Cryptology ePrint Archive, Paper 2016/714, 2016.

[Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Report 2021/582, 2021.

[STW25] Maximilian Schöffel, Hiandra Tomasi, and Norbert Wehn. Hw/sw implementation of mirith on embedded platforms. In *2025 IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)*, volume 1, pages 1–5, 2025.