Towards a Unified Memory-Less Framework for TCitH

Jesús-Javier Chi-Domínguez[1], Décio Luiz Gazzoni Filho[1,2], Marco Palumbi[1], and Luis Rivera-Zamarripa[1]

[1] Technology Innovation Institute, Abu Dhabi, UAE
{jesus.dominguez,decio.gazzoni,marco.palumbi,luis.zamarripa}@tii.ae
[2] Dept. of Electrical Engineering, State University of Londrina, Londrina, Brazil

Abstract. The current on-ramp NIST Competition for Additional Post-Quantum Digital Signature Schemes features two MPCitH variants: TCitH and VOLEitH. While VOLEitH yields shorter signatures and more stack memory, making it less suitable for constrained devices. In this work, we demonstrate that TCitH-based schemes are viable on embedded systems, such as Cortex-M4 devices. We present a simple, unified Zero-Knowledge Proof (ZKP) framework covering all TCitH-based submissions to the NIST competition. Our implementation achieves up to 99% reduction in stack usage over a baseline, with minimal code size overhead and negligible performance overhead. The framework is designed for extensibility: adding new schemes requires only implementing the mathematics of the underlying problem and the polynomial proof procedures. We further contribute a novel constant-time, bitsliced implementation of Rijndael-256 for embedded architectures, targeting the GGM tree Expand, PRG, and Commit functions central to TCitH-based schemes. This is an independent contribution of broader relevance, given NIST's ongoing standardization of Rijndael-256 and its use across MPCitH-based schemes. We believe that such a framework and implementation could help new developers and cryptographers when proposing new TCitH-based schemes.

Keywords: GGM Tree · MPCitH · TCitH · AES · Rijndael · Post-Quantum Cryptography

1 Introduction

The NIST on-ramp competition for additional post-quantum signatures included, in its second round, six constructions that rely on the Multi Party Computation in the Head (MPCitH) paradigm: FAEST [12], MQOM [13], SDitH [34], Mirath [4], PERK [1], and RYDE [9]. Recently, NIST announced that the first three advanced to the third round. There are two flavors for instantiating this family of schemes: Threshold Computation in the Head (TCitH) and Vector Oblivious Linear Evaluation in the Head (VOLEitH) frameworks. MQOM relies on the former, FAEST on the latter, and SDitH defines versions using both flavors. Both techniques

allow to model these signature protocols combining a Polynomial Interactive Oracle Proof (PIOP) with a Polynomial Commitment Scheme (PCS).

Despite the flexibility in describing and implementing TCitH and VOLEitH schemes through the PIOP formalism, existing approaches can be complex, particularly for non-experts, due to the lack of a unified framework. We address this by introducing a generic, easy-to-use framework that significantly simplifies implementation while remaining highly memory-efficient, making it well suited for embedded systems. In particular, our framework provides a clearer and more intuitive understanding of MQOM and SDitH constructions.

PIOP *and* PCS *Primitives [30,16]:* In a nutshell, a PIOP is a public-coin interactive proof system in which the prover sends polynomial oracles to the verifier, who queries them at chosen points and decides based on the returned evaluations. It is an information-theoretic abstraction whose soundness is independent of cryptographic assumptions. In contrast, a PCS is a cryptographic primitive that allows a prover to commit to a polynomial via a succinct string and later prove that it evaluates to a claimed value at a verifier-chosen point, while preventing equivocation between distinct polynomials. Rather, a PIOP specifies the proof system in an idealized oracle model, whereas a PCS realizes these oracles under standard cryptographic assumptions.

More precisely, a PIOP is a public-coin interactive protocol between a prover $\mathsf{P}(\mathsf{x},\mathsf{w})$ and a verifier $\mathsf{V}(\mathsf{x})$, where x denotes a statement and w a witness. The prover sends $t \in \mathbb{N}$ polynomial oracles $[\![\mathbf{p}_i]\!]_{i\in[t]}$ to the verifier for polynomials $\{\mathbf{p}_i\}_{i\in[t]}$, and the verifier responds with uniformly sampled challenge strings $\{\gamma_i\}_{i\in[t]}$. Next, the verifier derives t evaluation points $\{z_i\}_{i\in[t]}$ from challenges $\{\gamma_i\}_{i\in[t]}$, and queries to polynomial oracles $[\![\mathbf{p}_i]\!]_{i\in[t]}$. Once the oracles provide the polynomial evaluations

$$y_1 = \mathbf{p}_1(z_1), y_2 = \mathbf{p}_2(z_2), \dots \text{ and } y_t = \mathbf{p}_t(z_t),$$

the verifier outputs a decision bit to *accept* or *reject*. Whereas PCS is a primitive that allows P to commit to the polynomial $\mathbf{p}_i$ as some string com_i. Then, V requests com_i to be opened to the specific evaluation z_i of the underlying polynomial $\mathbf{p}_i$ through a Probabilistic Polynomial Time (PPT) protocol between P and V that interactively proves the statement $y_i = \mathbf{p}_i(z_i)$ and validates $\mathbf{p}_i(z_i)$ to be consistent with the given commitment com_i.

On the PIOP *formalism:* Fix a finite field $\mathbb{F}$ and a finite field extension $\mathbb{K}$ of $\mathbb{F}$. The frameworks of TCitH and VOLEitH fall into the scenario where P wants to convince V about the knowledge of a valid witness $\mathbf{x}$ satisfying $f(\mathbf{x}) = 0$ for some public degree-d polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$.

- The variant TCitH operates on Shamir Secret Sharing and employs two types of random vector polynomials. Let us denote by $P(\infty)$ the vector of leading coefficients of a vector polynomial $\mathbf{p} \in (\mathbb{F}[x])^n$. Let $N \in \mathbb{N}$ and $\phi\colon [N] \to \mathbb{K}$ be a public one-to-one function, where $[N]$ denotes $[1, N] \cap \mathbb{Z}$ and $\mathbb{K}$ is finite field extension of $\mathbb{F}$. Slightly abusing the notation, write $\mathbf{p}(i_\phi)$ to refer to P being evaluated at $i_\phi \coloneqq \phi(i)$. Then

- P distributes the secret witness $\mathbf{x}$ by sampling a random degree-1 vector polynomial $\mathbf{p}_{\mathbf{x}} \in (\mathbb{F}[x])^n$ such that $\mathbf{p}_{\mathbf{x}}(\infty) = \mathbf{x}$ and $\mathbf{p}_{\mathbf{x}}(i_\phi)$ defines the corresponding i^{th} share $[\![\mathbf{x}]\!]_i$ for each $i \in [N]$. Additionally, P samples a random degree-$(d-1)$ vector polynomial $\mathbf{p}_u \in (\mathbb{F}[x])^\eta$. Next, P sends the polynomial oracles $[\![\mathbf{p}_{\mathbf{x}}, \mathbf{p}_u]\!]$ to V.
- V responds with a random matrix $\Gamma \in \mathbb{K}^{\eta \times m}$.
- P reveals the vector polynomial $\mathbf{p}_\alpha = \mathbf{p}_u + \Gamma \cdot f(\mathbf{p}_{\mathbf{x}})$ to V.
- V samples a random element $z \in \mathbb{K}$ and queries $\{\mathbf{p}_{\mathbf{x}}(z), \mathbf{p}_u(z)\}$ from $[\![\mathbf{p}_{\mathbf{x}}, \mathbf{p}_u]\!]$. Next, it verifies that $\mathbf{p}_\alpha(z) = \mathbf{p}_u(z) + \Gamma \cdot f(\mathbf{p}_{\mathbf{x}}(z))$ holds.

The above PIOP is repeated τ times, with possible different values of $N = N_j \in \mathbb{N}$ per iteration, for some specified value of $\tau \in \mathbb{N}$ determined by the security requirements of each signature construction.

– The variant $\mathsf{VOLEitH}$ "merges" the τ polynomial oracles of the TCitH scenario into a single polynomial oracle that allows the opening of $L = \sum_{i \in [\tau]} N_i$ evaluations at the same time. This technique is deployed over correlations that distribute the 2-tuples $(\mathbf{u}, \mathbf{v}) \in \mathbb{F}^\ell \times \mathbb{K}^\ell$ to Prover and $(\mathbf{q}, \Delta) \in \mathbb{K}^\ell \times \mathbb{K}$ to Verifier, all chosen at random and subject to satisfying the correlation $\mathbf{q} = \mathbf{p}_{\mathbf{x}}(\Delta)$, where $\mathbf{p}_{\mathbf{x}}$ determines the degree-1 polynomial $x\mathbf{u} + \mathbf{v}$.

Tree Optimization Techniques: All of the aforementioned signature constructions employ the Goldreich-Goldwasser-Micali (GGM) tree technique to uniformly sample and commit polynomials $\{\mathbf{p}_{\mathbf{x}}, \mathbf{p}_u\}$, which are determined by $L = \sum_{e \in [\tau]} N_e$ random shares. Most of those signature schemes implement the Batch All-but-one Vector Commitment (BAVC) primitive that integrates the One Big Tree (OBT) optimization from [11], except for MQOM which opts for a forest approach of $\tau \in \mathbb{N}$ trees (for short, Forest_τ). Essentially, the OBT technique proposes building a binary tree with L leaves rather than working in a forest with τ trees such that the e^{th} tree has $N_e \in \mathbb{N}$ leaves. In particular, OBT determines a rearrangement of the leaves of the forest approach in an interleaved position. More precisely, the first τ leaves of the OBT determine the first leaf of the τ trees, the next leaves correspond to the second leaves, and so on. In terms of performance, both techniques essentially compute the same number of tree nodes and, hence, have the same running-time complexity. However, the main differences between the Forest_τ and OBT techniques include the following.

– OBT allows *shorter* signature sizes compared to the Forest_τ case; and
– OBT requires a *much bigger* GGM tree than in the Forest_τ scenario.

For example, the specifications of MQOM illustrate a friendly implementation on embedded platforms with low memory profiles. In contrast, the trade-off between performance and memory requirements is questionable for the current state-of-the-art with respect to OBT optimization since recomputation of the GGM tree is required [10]. [3] In summary, the size of the GGM trees highlights the stack memory bottleneck in both scenarios.

[3] A similar situation occurs for Forest_τ-based implementation on embedded platforms of MQOM [14], where some recomputation is required but not as heavy as for the OBT scenario.

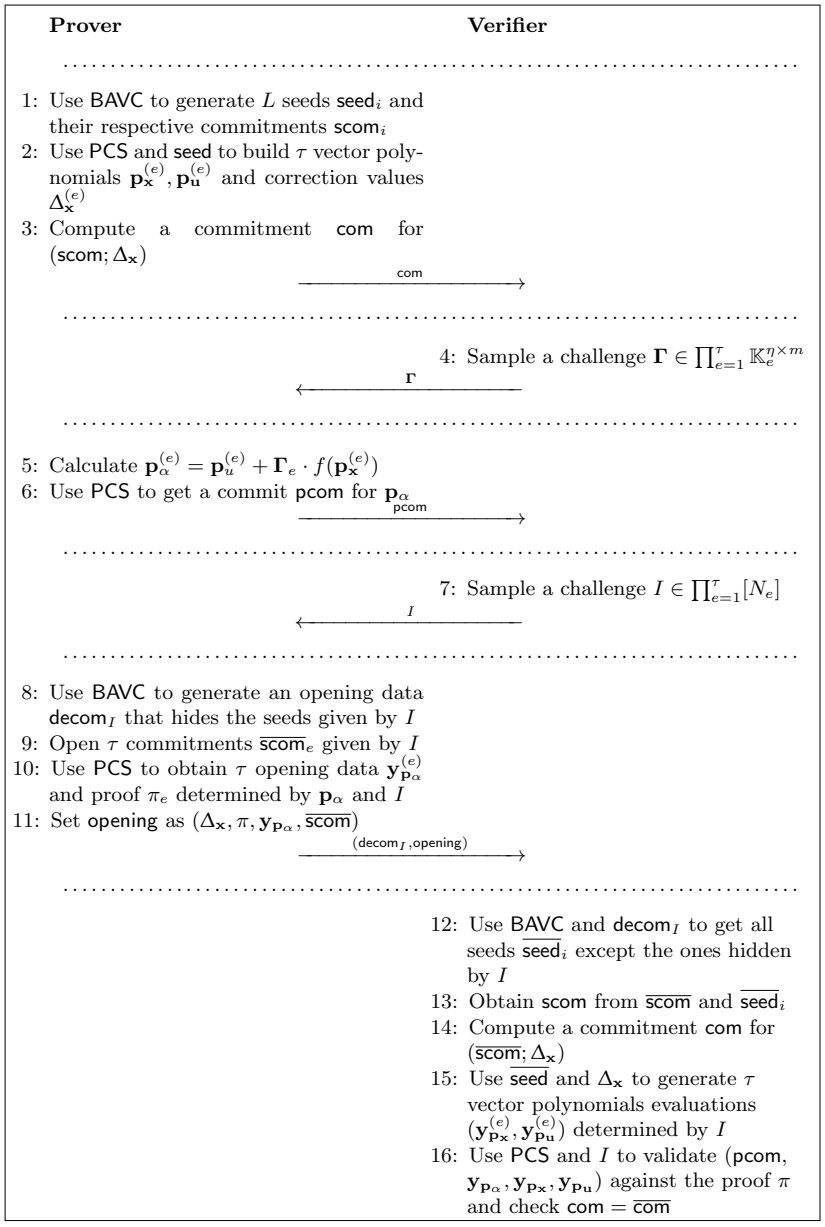


Fig. 1: High-level description of the baseline zero-knowledge proof based on the TCitH paradigm. In particular, $\mathbf{x}$ denotes a valid witness satisfying $f(\mathbf{x}) = 0$, for some public degree-d polynomial system relation $f \colon \mathbb{F}^n \to \mathbb{F}^m$.

1.1 Motivation

A single underlying Zero-Knowledge Proof (ZKP) can model these MPCitH-based signature schemes, but their specifications are complex to understand and do not reveal similarities between signature schemes (each construction has its own interpretation of how to describe the underlying ZKP on top of BAVC and PCS primitives). Understanding one signature scheme specification does not imply a straightforward understanding of the others – MPCitH-based signatures have an extra layer of complexity in what they are essentially proving, at least at the level of the signature and verification algorithms. Part of the complexity in comprehending the MPCitH-based signature constructions stems from how these two independently employed primitives, BAVC and PCS, split the body calculations. For experts in the field, this understanding is clear; however, for real-world implementers, it goes beyond their knowledge. To have a better picture, we provide in Figure 1 a high-level interpretation of the underlying ZKP for the TCitH-based signatures in [4,13,9,34], which we use as a baseline to be improved.

Remark 1. The current scheme proposals in [4,13,1,9,34] include additional optimizations; e.g., the index vector I is such that decom_{I_ψ} has a length bounded by a fixed threshold bound $\mathsf{T}_{\mathsf{open}}$, and its search is attached to a w-bit proof-of-work to reduce the number τ of parallel repetitions. For simplicity, we do not include these optimizations in Figure 1, but we must have in mind that the primitive BAVC is required $\mathcal{O}(2^w)$ times for optimized implementations of Figure 1.

To get an idea of the gap between theory and practice, one can look at real-world scenarios in which implementers have only a basic background in cryptography, leaving them open to security issues when they provide optimized code for calculations they do not understand. Hence, having a concrete, portable, and (as much as possible) simple protocol construction would help bridge the gap between cryptography in theory and practice. For instance, there is a simple high-level protocol framework for protocols such as Diffie-Hellman (DH) key exchange, the Digital Signature Algorithm (DSA), and Schnorr signatures, which can be instantiated with different underlying computational hard problem assumptions (e.g., discrete logarithms over a finite field or an elliptic curve). Taking into account everything discussed before, our primary motivation relies on the following question.

> *Can we describe each* MPCitH*-based signature scheme in [12,4,13,1,9,34] through a simple and unified* ZKP *framework that is easy to understand and implement?*

1.2 Contribution

We affirmatively answer the above question and propose a unified ZKP framework that allows us to describe each TCitH-based signature scheme in [4,13,9,34], while still allowing non-experts in the field to understand the specific differences

and similarities between the signature constructions. We present our unified primitive and ZKP in Section 3, respectively.

Our proposal allows us to encapsulate the main built-in algorithms required to describe our ZKP framework in a compact and simpler way. We emphasize that we are not proposing any new ZKP, and our results must be seen as a reformulation and understanding of the baseline ZKP of each TCitH-based signature scheme in [4,13,9,34]. In addition to the unified theoretical description, we significantly reduce memory complexity for OBT-related calculations, from linear to logarithmic in L; making the implementation of these schemes as friendly as possible for resource-constrained embedded devices. In particular, our simplified ZKP framework helps us to spot new stack memory bottlenecks, now given by the vector polynomials $\mathbf{p_x}$ within a size determined by the bit-length of the witness. We detail in Section 4 each built-in algorithm required in our ZKP framework.

We estimate the stack memory requirements for each scheme in [4,13,9,34], and discuss further implications of our work to enable efficient implementations that run in embedded devices with limited resources. Moreover, when plugging our stack memory analysis in MQOM, our estimates hint at smaller stack memory usages through our proposal compared to [14]. We stress that our proposal enables efficient implementations where there are no additional re-computations related to the OBT and polynomial calculations, while the most compact implementation in [14] requires some additional re-computations. For SDitH, we provide a detailed discussion of the main constraints in stack memory requirements. We estimate the stack memory requirements for each scheme in Section 4.1.

We provide experiments in Cortex-M4 devices showing the practicability and efficiency of our proposal. We center our experiments on the schemes in [4,9] since they provide a public implementation of their TCitH variant and do not count with an implementation in Cortex-M4, and we validate our stack memory estimates; however, we highlight that our results also apply to [13,12]. We provide a Proof-of-Concept (PoC) implementation available at [21], which illustrates that our proposal does not add any slowdown in performance. Our implementation is as optimized as possible at the level of the ZKP framework. We consider the reference implementation from [4,9]; hence, providing an optimized implementation for the arithmetic related to the underlying problem will provide faster timings. Even so, our results illustrate smaller stack requirements than the implementation of FAEST in [10], and illustrate decent performance for a reference implementation at the level of the arithmetic layer. We present our experiments and comparisons with the state-of-the-art in Section 6.

Additionally, we provide a constant-time optimized implementation of AES-128 and Rijndael-256 for embedded architectures. Our experiments show that we significantly advance the state of the art in embedded Rijndael-256 implementations, particularly for the Cortex-M4: our implementation of Rijndael-256 is around 2.6 times faster than the optimized implementation in [14]. Hence, our improvements related to AES/Rijndael could be of interest to MQOM and SDitH teams, since their main calculations also rely on these encryption schemes.

We point out that our proposal differs sufficiently from the original implementations to break compatibility with the provided KATs. This does not affect security: the main difference lies in the order of commitment absorption. Since the schemes are still being considered and updating the KATs would have minimal impact, we believe that our findings could be of interest to MQOM and SDitH teams. In particular, there is no implementation of the TCitH variant of SDitH, so there are no KATs to be broken and SDitH could consider our findings when providing a variant implementation of TCitH. On the other hand, given the current status of the security proof (in the random oracle model) of MQOM from [31,24,32], the MQOM team is likely to update their KATs according to the security proof requirements; Thus, they could consider our findings to provide a more compact and faster implementation with reduced stack memory usage.

Still, our ZKP framework and its implementation offer lessons and motivate efficient implementations in both stack memory and performance efficiency. We believe that such a framework and implementation could help new developers and cryptographers when proposing new TCitH-based schemes. For example, our implementation is as generic as possible, so adding new constructions requires only implementing the mathematics of the underlying problem and the polynomial proof procedures. We suggest specific directions for future work in Section 7, with a focus on MQOM and SDitH.

Remark 2. We only consider TCitH-based signatures, since VOLEitH-based constructions in [12,1,34] include merging and consistency checks for the VOLEitH correlations as additional steps. Even if those extra steps do not add any other "calls" to BAVC or PCS primitives, the consistency check steps require a sufficiently large matrix over a large field of at least 2^λ elements with λ being the security parameter. TCitH-based signatures are expected to require much less stack memory than VOLEitH, making them attractive for constrained devices.

2 Preliminaries

Through the manuscript, we use $\top$ and $\bot$ to denote the truth values "True" and "False", respectively. For any positive integer $t \in \mathbb{N}$, we denote by $[t]$ (resp. $[t-1]_0$) the list of integers in $[1,t]$ (resp. $[0,t-1]$). Let $\mathbb{F}$ be a finite field, and fix integers $n, m, \eta \in \mathbb{N}$ along with a security parameter $\lambda \in \mathbb{N}$. Consider a witness space $\mathcal{W}$ of vectors $\mathbf{x} \in \mathbb{F}^n$ satisfying $f(\mathbf{x}) = 0$, for some public degree-d polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$.

Let $\mathsf{PRG}\colon \{0,1\}^\lambda \to \mathbb{F}^n \times \mathbb{F}^\eta$ that on input $\mathsf{s} \in \{0,1\}^\lambda$ it outputs a pseudo-random element $(\mathbf{x}', \mathbf{u}')$ of $\mathbb{F}^n \times \mathbb{F}^\eta$, and let $\mathsf{Expand}\colon \{0,1\}^\lambda \to \{0,1\}^\lambda \times \{0,1\}^\lambda$ be a function that on input $\mathsf{s} \in \{0,1\}^\lambda$ outputs two λ-bit strings s_0 and s_1. We assume $\mathsf{Commit}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ defines a commitment function that on input $\mathsf{s} \in \{0,1\}^\lambda$ outputs a 2λ-bit string com. In addition, we model a commitment scheme Committer as a stateful interactive object that supports incremental absorption of messages followed by finalization. That is, Committer is composed of the following three Probabilistic Polynomial Time (PPT) algorithms, each algorithm with access to a random oracle and the security parameter λ:

$\mathsf{Init}()$: Initializes a private internal state $s \in \{0,1\}^{\bullet\lambda}$ of fixed length $\mathcal{O}(\lambda)$.
$\mathsf{Absorb}(m)$: Given a message $m \in \{0,1\}^*$, absorbs and updates the private internal state s from the usage of m and the current internal state s.
$\mathsf{Finalize}() \to \mathsf{com}$: Outputs a commitment $\mathsf{com} \in \{0,1\}^{2\lambda}$ determined by the private internal state s.

We highlight that Commit can define a different commitment compared to $\mathsf{Committer}$ [4,1,9,34], but in some scenarios it happens that $\mathsf{com} = \mathsf{Commit}(m)$ is equivalent to sequentially performing $\mathsf{Committer.Init}()$, $\mathsf{Committer.Absorb}(m)$ and $\mathsf{com} = \mathsf{Committer.Finalize}()$. [4]

On the other hand, we consider collections of $\tau \in \mathbb{N}$ vectors such that the e^{th} vector has length $N_e \in \mathbb{N}$, and write $L = \sum_{e\in[\tau]} N_e$. We let $\psi\colon (\{1\} \times [N_1]) \cup \cdots \cup (\{\tau\} \times [N_\tau]) \to [L]$ be the bijection that determines the interleaved rearrangement of L given strings $\mathsf{s}_{L-1}, \ldots, \mathsf{s}_{2L-1} \in \{0,1\}^\lambda$, and $\phi_e\colon [N_e] \to \mathbb{K}_e$ be an embedded function of $[N_e]$ in a finite field $\mathbb{K}_e \supset \mathbb{F}$ with $\#\mathbb{K}_e \geq N_e$.

GGM *Tree:* Given a λ-bit string s_0, the GGM tree technique allows generating L λ-bit strings $\mathsf{s}_{L-1}, \ldots, \mathsf{s}_{2L-1} \in \{0,1\}^\lambda$ by constructing a binary tree with root s_0 and nodes of the form $(\mathsf{s}_{2j+1}, \mathsf{s}_{2j+2}) = \mathsf{Expand}(\mathsf{s}_j)$ for each $i \in [L-2]$. The nodes s_{2j+1} and s_{2j+2} correspond to the children of the parent node s_j. Consequently, the last L nodes $\mathsf{s}_{L-1}, \ldots, \mathsf{s}_{2L-1}$ determine the leaves of the tree. By construction, the GGM tree technique follows a Breadth-First Search (BFS) procedure, storing $2L-1$ nodes in total and imposing a specific ordering on the calculation of the strings $\mathsf{s}_{L-1}, \ldots, \mathsf{s}_{2L-1}$. Figure 2 illustrates the structure of the GGM tree.

Additional notation on element subscripts: To avoid any possible confusion, we represent elements in $[\tau]$ by letter e (with possible sub-indexes such as e_j), while for $[N_e]$ we choose i and i_j.

Remark 3. Current constructions as in [4,1,9,34] implement the functions PRG, Expand and Commit internally taking into account a public random string $\mathsf{salt} \in \{0,1\}^{2\lambda}$, the node identifier $j \in [2N-1]$ of the input seed s_j, and $\psi^{-1}(j) := (e_j, i_j) \in [\tau] \times [N_{e_j}]$. We consider these facts, and assume that the public random string salt is included in both decom and decom_{I_ψ}. We highlight the dependency of salt on $\mathsf{fun} \in \{\mathsf{Expand}, \mathsf{PRG}, \mathsf{Commit}\}$ by writing $\mathsf{fun}(\mathsf{s}; \mathsf{salt})$ instead of $\mathsf{fun}(\mathsf{s})$.

3 On the Zero-Knowledge Proof from a Unified Primitive

Although it is true that Figure 1 provides a high-level description encapsulating the PIOP formalism of the constructions in [4,1,9,34] at the level of the building-algorithms for BAVC and PCS, it also allows us to spot some caveats regarding large stack memory requirements. We detail these caveats next. In

[4] An example of this modeling for Commit and $\mathsf{Committer}$ is the family of $\mathsf{SHA3}$ hash functions, which is commonly used in practice.

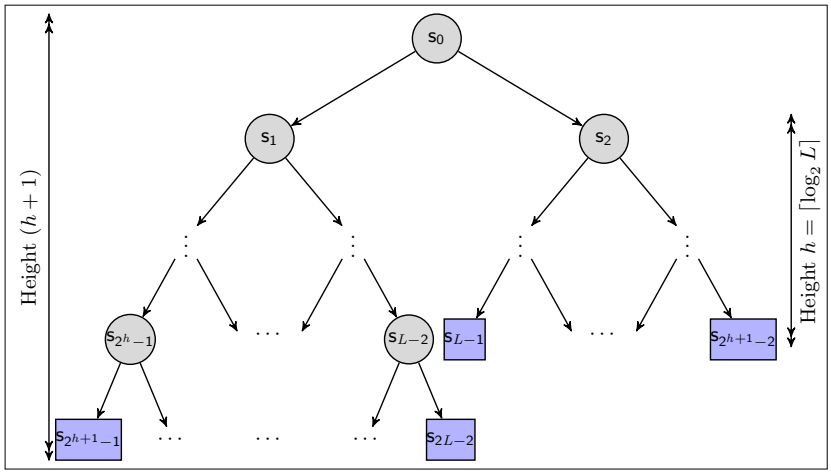


Fig. 2: Breadth-First Search (BFS) technique: One Big Tree (OBT) with $L = \sum_{i\in[\tau]} N_i$ leaves where $N_i \in \mathbb{N}$ and $\tau \in \mathbb{Z}$ are a positive integer.

practice, Line 1 in Figure 1 considers that BAVC returns strings of the form $\mathsf{decom} = \{\mathsf{decom}_i \in \{0,1\}^\lambda\}_{i\in[2L-1]}$, $\mathsf{scom} = \{\mathsf{scom}_i \in \{0,1\}^{2\lambda}\}_{i\in[L]}$ and $\mathsf{seed} = \{\mathsf{seed}_i \in \{0,1\}^\lambda\}_{i\in[L]}$. The string decom determines a GGM tree with L leaves being seed and scom coincides with the commitment of the seeds, which gives a stack memory requirement of $(4L-1)\lambda$ bits. In addition, these three strings play a crucial role in the subsequent steps outlined in Figure 1. More precisely,

- Line 2 requires L seeds $\{\mathsf{seed}_i\}_{i\in[L]}$;
- Line 3 requires L commits $\{\mathsf{scom}_i\}_{i\in[L]}$;
- Line 8 requires the entire GGM tree with leaves $\{\mathsf{seed}_i\}_{i\in[L]}$;
- Line 9 requires the τ commits $\{\mathsf{scom}_i\}$ determined by I;
- Line 12 calculates $(L-\tau) \approx L$ seeds, excluding the ones hidden by I;
- Line 13 calculates $(L-\tau) \approx L$ commits corresponding to the seeds in Line 12;
- Line 14 requires L commits $\{\mathsf{scom}_i\}_{i\in[L]}$;

Indeed, the storage of $(\mathsf{scom}, \mathsf{decom}, \mathsf{seed})$ can be omitted by taking decom as the root of the GGM tree and considering the option of recalculating scom and seed at each required step, implying a considerable slowdown in performance: seed and scom are needed twice for Prover and Verifier, while decom is required once and only for Prover in Figure 1. However, for optimized implementations decom must be called $\mathcal{O}(2^w)$ times as highlighted in Remark 1, and the calculation of polynomials $\{\Delta_{\mathbf{x}}^{(e)}, \mathbf{p}_{\mathbf{x}}^{(e)}, \mathbf{p}_{\mathbf{u}}^{(e)}\}_{e\in[\tau]}$ in Line 2 processes seeds according to a predetermined interleaved function ψ: $(\{1\} \times [N_1]) \cup \cdots \cup (\{\tau\} \times [N_\tau]) \to [L]$ from $i_1 \in \{1\} \times [N_1]$ to $i_L \in \{\tau\} \times [N_\tau]$ in lexicographic ordering, that is, it starts at $\mathsf{seed}_{\psi(i_1)}$ and ends at $\mathsf{seed}_{\psi(i_L)}$ instead of going from the first seed seed_1 to the last seed seed_L. Then, the interleaved approach to seed processing with $\mathsf{decom} \in \{0,1\}^\lambda$ as the root of the GGM tree would require the recomputation of $\tilde{\mathcal{O}}(L)$ nodes with a considerable impact on performance.

As a solution to reduce stack memory requirements without a significant slowdown in performance, we propose a primitive that merges and encapsulates both building-algorithms of the primitives BAVC and PCS. Our proposal choice relies on the following main points:

- The computations in Lines 1 to 3 in Figure 1 can be merged by allowing an interleaved processing of the polynomials $\{\Delta_{\mathbf{x}}^{(e)}, \mathbf{p}_{\mathbf{x}}^{(e)}, \mathbf{p}_{\mathbf{u}}^{(e)}\}_{e\in[\tau]}$ along with seeds $\{\mathsf{seed}_i\}_{i\in[L]}$ and commitment $\{\mathsf{scom}_i\}_{i\in[L]}$. Similarly, Lines 12 and 13 can be merged.
- The calculation of commitment com in Line 3 of Figure 1 does not need to store the full set of commitments $\{\mathsf{scom}_i\}_{i\in[L]}$; it can be done on the fly in practice. This observation also applies to Line 13. Consequently, the τ commitments $\{\mathsf{scom}_e\}_{i\in[\tau]}$ in Line 9 should be obtained from scratch, adding a small overhead in performance since $\tau = \mathcal{O}(\log_2 L)$ is usually taken in practice.
- The length of decom_I can be determined by only employing $I_\psi = \{\psi(e, i_e)\}_{i\in I}$ and without the requirement of decom. In particular, the algorithm for opening in Line 8 can be split into two algorithms Unfold and Open, where algorithm Unfold outputs the indexes $\{i_j\}$ of decom_{I_ψ} and Open returns decom_{I_ψ}.

3.1 Batch All-but-one Polynomial Commitments

We present our proposed primitive in Definition 1. Our primitive benefits from being instantiated and implemented through polynomial time algorithms in the variable L and with a logarithmic memory complexity in L.

Definition 1 (Batch All-but-one Polynomial Commitment Scheme). *Let $\lambda \in \mathbb{N}$ be the security parameter and consider $\tau \in \mathbb{N}$ vectors such that the e^{th} vector has length $N_e \in \mathbb{N}$, and write $L = \sum_{e\in[\tau]} N_e$. Let $\mathcal{M} = \{0,1\}^\lambda$ be the message space. Let us denote by i_e the index of the e^{th} vector that remains unopened and by $I \in [N_1] \times \cdots \times [N_\tau]$ the vector $(i_1, \ldots, i_\tau)$ comprising all the indices that remain unopened. Let $\psi\colon (\{1\} \times [N_1]) \cup \cdots \cup (\{\tau\} \times [N_\tau]) \to [L]$ be the bijection that determines the interleaved position of the messages, and let $\phi_e\colon [N_e] \to \mathbb{K}_e$ be an embedded function of $[N_e]$ in $\mathbb{K}_e \supset \mathbb{F}$ with $\#\mathbb{K}_e \geq N_e$. A (non-interactive) Batch All-but-one Polynomial Commitment scheme* BAPC *in the Random Oracle model is composed of the following six Probabilistic Polynomial Time (PPT) algorithms, each algorithm with access to a random oracle and the security parameter λ,*

$\mathsf{Build}(\mathbf{x}) \to (\mathsf{com}, \mathsf{decom}, \{\Delta_{\mathbf{x}}, \mathbf{p}_{\mathbf{x}}, \mathbf{p}_{\mathbf{u}}\})$: *On input a witness $\mathbf{x}$ satisfying $f(\mathbf{x}) = 0$, for some public degree-d polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$, outputs a commitment $\mathsf{com} \in \{0,1\}^{2\lambda}$ with opening information $\mathsf{decom} \in \{0,1\}^{\bullet\lambda}$ for*

messages $(m_1^{(e)}, \ldots, m_{N_e}^{(e)})_{e\in[\tau]} \in \mathcal{M}^{N_1} \times \cdots \times \mathcal{M}^{N_\tau}$*, and vector polynomials*

$$\begin{aligned}
\Delta_{\mathbf{x}} &= \{\Delta_{\mathbf{x}}^{(e)} = \mathbf{x} - \sum\nolimits_i [\![\mathbf{x}^{(e)}]\!]_i\}_{e\in[\tau]},\\
\mathbf{p}_{\mathbf{x}} &= \{\mathbf{p}_{\mathbf{x}}^{(e)}\}_{e\in[\tau]} \textit{ such that } \mathbf{p}_{\mathbf{x}}^{(e)}(\infty) = \mathbf{x}, \textit{ and}\\
\mathbf{p}_{\mathbf{u}} &= \{\mathbf{p}_{\mathbf{u}}^{(e)}\}_{e\in[\tau]},
\end{aligned}$$

where $\mathbf{p}_{\mathbf{x}}^{(e)}$ *is a degree-d vector polynomial and* $\mathbf{p}_{\mathbf{x}}^{(e)}(i_\phi)$ *defines the corresponding* i^{th} *random share vector* $[\![\mathbf{x}^{(e)}]\!]_i$ *seeded with message* $m_i^{(e)}$*, and* $\mathbf{p}_{\mathbf{u}}^{(e)}$ *is a random degree-*$(d-1)$ *vector polynomial* $\mathbf{p}_{\mathbf{u}} \in (\mathbb{F}[x])^\eta$ *fixed by* $(m_1^{(e)}, \ldots, m_{N_e}^{(e)})$*.*

$\mathsf{Commit}(\mathbf{p}_\alpha) \to \mathsf{pcom}$: *On input a collection of* τ *degree-*$(d-1)$ *vector polynomials* $\mathbf{p}_\alpha^{(e)} = \{\mathbf{p}_\alpha^{(e)} \in \mathbb{K}_e[x] \mid \mathbf{p}_\alpha = \mathbf{p}_{\mathbf{u}}^{(e)} + \mathbf{\Gamma}_e \cdot f(\mathbf{p}_{\mathbf{x}}^{(e)})$ *for some* $\mathbf{\Gamma}_e \in \mathbb{K}_e^{\eta\times m}\}_{e\in[\tau]}$*, outputs a commitment* $\mathsf{pcom} \in \{0,1\}^{2\lambda}$*.*

$\mathsf{Unfold}(I) \to \mathsf{idcom}_{I_\psi} \vee \perp$: *On input an index vector* I*, either outputs an identifier vector* $\mathsf{idcom}_{I_\psi} \in [2L]^\bullet$ *determining an opening information* $\mathsf{decom}_{I_\psi} \in \{0,1\}^{\bullet\lambda}$ *for* $I_\psi = (\psi(1,i_1), \ldots, \psi(\tau,i_\tau))$ *or* $\perp$*.*

$\mathsf{Open}(\mathsf{decom}, \mathsf{idcom}_{I_\psi}, I, \mathbf{p}_\alpha) \to (\mathsf{decom}_{I_\psi}, \mathbf{y}_{\mathbf{p}_\alpha}, \pi, \mathsf{opcom})$: *On input an opening information* decom*, an identifier vector* idcom_{I_ψ} *along with the index vector* I *itself, and a collection of* τ *degree-*$(d-1)$ *vector polynomials* $\mathbf{p}_\alpha$*, outputs an opening information* $\mathsf{decom}_{I_\psi} \in \{0,1\}^{\bullet\lambda}$ *for* I_ψ*, a collection of* τ *vector polynomial evaluations* $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)} = \mathbf{p}_\alpha^{(e)}(\phi_e(i_e))$ *with a non-interactive proof of evaluation* π*, and* τ *opened commitments* $\mathsf{opcom} = \{\mathsf{Commit}(m_{\psi(i_e)}^{(e)})\}_{e\in[\tau]}$*.*

$\mathsf{Rebuild}(\mathsf{opcom}, \mathsf{decom}_{I_\psi}, I, \Delta_{\mathbf{x}}) \to (\mathsf{com}, \{\mathbf{y}_{\mathbf{p}_{\mathbf{x}}}, \mathbf{y}_{\mathbf{p}_{\mathbf{u}}}\}) \vee \perp$: *On input a collection of* τ *opened commitments, an opening information* decom_{I_ψ} *for an index vector* I *and the index vector itself along with the correction values* $\Delta_{\mathbf{x}}$*, either accepts the opening and outputs the reconstructed commitment* com *and collection of* τ *vector polynomial evaluations* $\mathbf{y}_{\mathbf{p}_{\mathbf{x}}}^{(e)} = \mathbf{p}_{\mathbf{x}}^{(e)}(\phi_e(i_e))$ *and* $\mathbf{y}_{\mathbf{p}_{\mathbf{u}}}^{(e)} = \mathbf{p}_{\mathbf{u}}^{(e)}(\phi_e(i_e))$*, or rejects the opening and outputs* $\perp$*.*

$\mathsf{Verify}(\mathsf{pcom}, \mathbf{y}_{\mathbf{p}_\alpha}, \mathbf{y}_{\mathbf{p}_{\mathbf{x}}}, \mathbf{y}_{\mathbf{p}_{\mathbf{u}}}, \pi, I) \to \top \vee \perp$: *On input a commitment* pcom*, and a claimed collection of* τ *vector polynomial evaluations* $\mathbf{y}_{\mathbf{p}_\alpha}$ *along with the proof of evaluation* π *and vector polynomials evaluations* $\mathbf{y}_{\mathbf{p}_{\mathbf{x}}}$ *and* $\mathbf{y}_{\mathbf{p}_{\mathbf{u}}}$*, it verifies the proof* π *against the claim* $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)} = \mathbf{p}_\alpha^{(e)}(\phi_e(i_e))$ *and validates* $\mathbf{p}_\alpha^{(e)}(\phi_e(i_e))$ *is consistent with the polynomial commitment* pcom*; in that case, it returns* $\top$*. Otherwise, it returns* $\perp$*.*

3.2 A Simplified Zero-Knowledge Proof Through BAPC Primitive

At first sight, our proposed primitive in Definition 1 looks complex and counterintuitive. However, we next explain each built-in algorithm in Definition 1 in general terms and illustrate the main advantages of this proposed primitive in describing a Zero-Knowledge Proof (ZKP) protocol that unifies the constructions in [4,13,9,34].

Let $\mathbf{x}$ be a valid witness that satisfies $f(\mathbf{x}) = 0$, for some public degree-d polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$. Consider the scenario where a prover

P wants to convince a verifier V of the knowledge of $\mathbf{x}$. We present the aforementioned simplified ZKP protocol in Figure 3, and summarize each step below.

- In Line 1 in Figure 3, P employs BAPC.Build to calculate the commitment com of all seeds $\{\mathsf{seed}_i\}_{i\in[L]}$ (i.e., GGM tree leaves) required to sample and generate vector polynomials $\mathbf{p_x}$ and $\mathbf{p_u}$. The main purpose of the BAPC.Build algorithm is to output com, decom and the vector polynomials $\mathbf{p_x}$ and $\mathbf{p_u}$ along with correction values $\Delta_{\mathbf{x}}$. The latter values of $\Delta_{\mathbf{x}}$ aim to help V to recompute com and determine polynomial evaluations for $\mathbf{p_x}$ and $\mathbf{p_u}$. Hence, P sends $(\mathsf{com}, \Delta_{\mathbf{x}})$ to V.
- In Line 2 of Figure 3, V samples a random challenge matrix $\mathbf{\Gamma} \in \mathbb{K}_1^{\eta\times m} \times \cdots \times \mathbb{K}_\tau^{\eta\times m}$ and send it to P.
- In Lines 3 and 4 of Figure 3, P calculates the vector polynomial $\mathbf{p}_\alpha = \{\mathbf{p}_{\mathbf{u}}^{(e)} + \mathbf{\Gamma}_e \cdot f(\mathbf{p}_{\mathbf{x}}^{(e)})\}_{e\in[\tau]}$ and calls BAPC.Commit to compute the polynomial commitment pcom. Then P sends pcom to V.
- In Line 5 of Figure 3, V samples a random challenge index vector $I = (i_1, \ldots, i_\tau)$, and send it to P.
- In Line 6 of Figure 3, P uses BAPC.Unfold to determine the indexes of the opening information for I_ψ. While in Line 7, P calls BAPC.Open to calculate the polynomial evaluations $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)} = \mathbf{p}_\alpha^{(e)}(\phi_e(i_e))$, a non-interactive proof of evaluation π, the opened commitments opcom and the opening information decom_{I_ψ} for I_ψ. [5] Then, P sends $(\mathsf{decom}_{I_\psi}, \mathbf{y}_{\mathbf{p}_\alpha}, \pi, \mathsf{opcom})$ to V.
- In Line 8 of Figure 3, V calls BAPC.Rebuild to compute the commitment $\overline{\mathsf{com}}$ along with polynomial evaluations $\mathbf{y}_{\mathbf{p_x}}^{(e)} = \mathbf{p}_{\mathbf{x}}^{(e)}(\phi_e(i_e))$ and $\mathbf{y}_{\mathbf{p_u}}^{(e)} = \mathbf{p}_{\mathbf{u}}^{(e)}(\phi_e(i_e))$.
- In Line 9 of Figure 3, V validates com is equal to $\overline{\mathsf{com}}$ and that BAPC.Verify accepts against the claim $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)} = \mathbf{p}_{\mathbf{u}}^{(e)}(\phi_e(i_e)) + \mathbf{\Gamma}_e \cdot f(\mathbf{p}_{\mathbf{x}}^{(e)}(\phi_e(i_e)))$ and checks $\mathbf{p}_\alpha^{(e)}(\phi_e(i_e))$ is consistent with the polynomial commitment pcom. [6]

4 A TCitH-based instantiation of Figure 3

As previously highlighted, the GGM tree technique comes with a BFS-like procedure that forces the storage of a full binary tree with $2L-1$ nodes. To reduce the number of tree nodes to be stored, we suggest a Depth-First-Search (DFS) approach to process the same set of λ-bit strings $\mathsf{s}_0, \ldots, \mathsf{s}_{2L-1}$. In particular, the DFS-like procedure computes the tree leaves starting at $\mathsf{s}_{2^{h+1}-1}$ and ending at $\mathsf{s}_{2^{h+1}-2}$. More precisely, the DFS-like procedure calculates the list of leaves as

$$\mathsf{Leaves}' = \left[\mathsf{s}_{2^{h+1}-1}, \ldots \mathsf{s}_{2L-2}, \mathsf{s}_{L-1}, \ldots \mathsf{s}_{2^{h+1}-2}\right],$$

[5] π describes a PPT algorithm that validates $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)} = \mathbf{p}_{\mathbf{u}}^{(e)}(\phi_e(i_e)) + \mathbf{\Gamma}_e \cdot f(\mathbf{p}_{\mathbf{x}}^{(e)}(\phi_e(i_e)))$.

[6] From $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)}$, $\mathbf{y}_{\mathbf{p_x}}^{(e)}$, $\mathbf{y}_{\mathbf{p_u}}^{(e)}$ and $\mathbf{\Gamma}$, V is assumed to have enough information to reconstruct the degree-$(d-1)$ vector polynomial $\mathbf{p}_\alpha$ via polynomial interpolation, without revealing the degree-d vector polynomial $\mathbf{p_x}$. In practice, $\mathbf{y}_{\mathbf{p}_\alpha}^{(e)}$ encodes $d-1$ elements that determine the evaluation point and additional data for the proof.

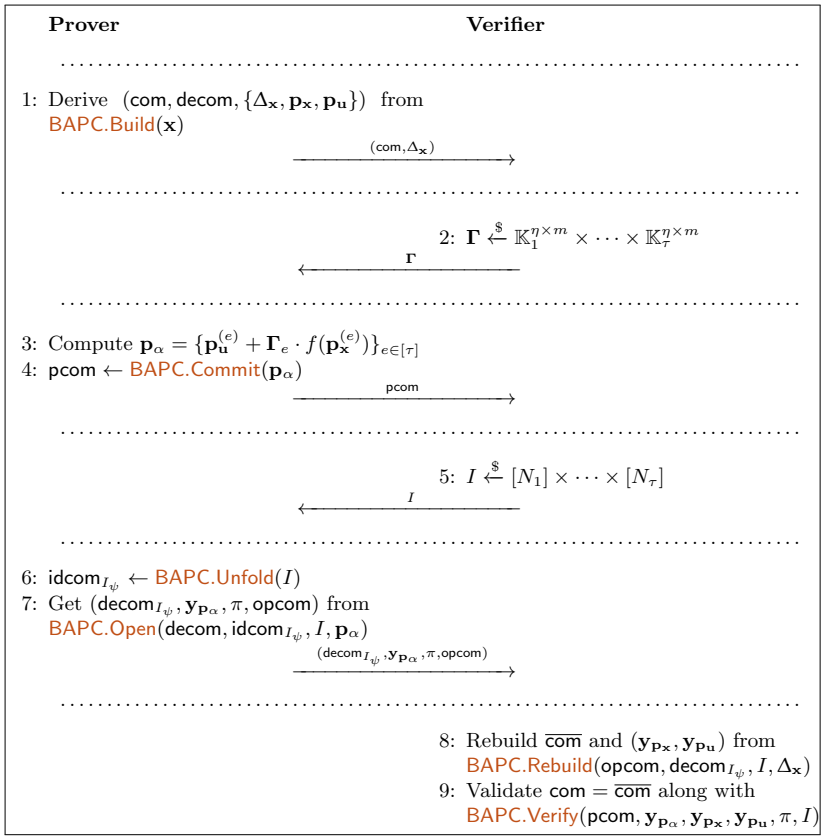


Fig. 3: This work: Zero-knowledge proof through BAPC primitive. It follows the notation from Definition 1. In particular, $\mathbf{x}$ denotes a valid witness satisfying $f(\mathbf{x}) = 0$, for some public degree-d polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$.

while the original BFS-like procedure considers the list of leave as

$$\mathsf{Leaves} = \left[\mathsf{s}_{L-1}, \ldots \mathsf{s}_{2^{h+1}-2}, \mathsf{s}_{2^{h+1}-1}, \ldots \mathsf{s}_{2L-2}\right].$$

We state in Lemma 1 this permutation equivalence, which allows us to argue why our choices determine the same outputs (up to a permutation) as computed in the baseline framework in Figure 1.

Lemma 1. *Consider* $L = \sum_{i \in [\tau]} N_i$ *for some positive integers* $N_i \in \mathbb{N}$ *and* $\tau \in \mathbb{Z}$*, and let* $\sigma\colon [L]_0 \to [L]_0$ *be the function given by* $x \mapsto (2^h + x) \bmod L$ *where* $h = \lceil \log_2 L \rceil$*. Let* Leaves *and* Leaves′ *be the* GGM *tree leaves obtained from the* BFS *and* DFS *procedures, respectively. Then*

$$\mathsf{Leaves}' = \left[\mathsf{Leaves}_{\sigma(0)}, \mathsf{Leaves}_{\sigma(1)}, \ldots \mathsf{Leaves}_{\sigma(L-2)}, \mathsf{Leaves}_{\sigma(L-1)}\right].$$

Proof. It follows from the ordering given in the lists Leaves and Leaves′. □

In addition to the permutation equivalence between the list of GGM tree leaves for both BFS and DFS techniques, we need an explicit mapping relation between how each node is processed. In particular, we require a node subindex labeling that allows efficient processing of each GGM tree node with an optimal stack memory requirement, while still being compatible with the BFS search, so that Lemma 1 works straightforwardly. For that purpose, we describe the labeling of each node below.

Label of nodes: As illustrated in Figure 2, the first node corresponds to the root s_0 of the GGM tree. The first and second children of the parent node s_0 determine the second and third nodes s_1 and s_2, and so on. By construction, the children of each parent node being at the same height level determine consecutive nodes in the BFS-like setting, while the DFS technique processes each GGM tree node in a different ordering; for instance, it process each node through a path starting from the root s_0 and ending at a certain node s_j (such a path is usually encoded as a bit-string $\mathsf{b} \in \{0,1\}^{\leq h}$). [7] To highlight the different processing of GGM tree nodes between these two approaches, we denote GGM tree nodes computed in the DFS method by letter z. Therefore, to allow both BFS and DFS methods to calculate the same set of nodes, we are implicitly required to find a bit-string encoding $\iota(j) \in \{0,1\}^{\leq h}$ for the subindex j of the j^{th} node s_j such that $\mathsf{z}_{\iota(j)} = \mathsf{s}_j$ for each $j \in [2L-1]_0$. We detail in Lemma 2 the corresponding bit-string encoding function ι.

Lemma 2. *Consider* $L = \sum_{i\in[\tau]} N_i$ *for some positive integers* $N_i \in \mathbb{N}$ *and* $\tau \in \mathbb{Z}$, *and* $\mathcal{Z}_L = \{\varnothing\} \cup \{\mathsf{b} \in \{0,1\}^{\leq h} \mid \mathsf{int}(\mathsf{b}) \in [2L-1]_0\}$ *where* $h = \lceil \log_2 L \rceil + 1$. *Let* $\iota^{-1} \colon \mathcal{Z}_L \to [2L-1]_0$ *be the function given by the relation*

$$\iota^{-1} \colon \mathsf{b} \mapsto \begin{cases} 0 & \text{if } \mathsf{b} \text{ is } \varnothing, \\ (2^{\#\mathsf{b}} \oplus \mathsf{int}(\mathsf{b})) - 1 & \text{otherwise.} \end{cases}$$

The function ι *describes a bijection between the lists of* GGM *tree nodes* $[\mathsf{s}_j \mid j \in [2L-1]_0]$ *and* $[\mathsf{z}_\mathsf{b} \mid \mathsf{b} \in \mathcal{Z}_L]$ *computed in the* BFS *and* DFS *procedures. In particular,* ι^{-1} *satisfies* $\mathsf{z}_\mathsf{b} = \mathsf{s}_{\iota^{-1}(\mathsf{b})}$ *for each* $\mathsf{b} \in \mathcal{Z}_L$.

Proof. By construction, ι^{-1} defines a bijection since it is injective and $\#\mathcal{Z}_L = \#[2L-1]_0 = 2L-1$. □

As the main implication of Lemmas 1 and 2, observe that our the ZKP protocol in Figure 3 computes exactly the same set of leaves (and hence same commitments per leaf) than Figure 1, implying that the vector polynomials in both ZKP protocols are equal. However, the commitment com differs in both protocols; in particular, our proposal is computing com as the commitment of $\{\mathsf{com}_{\iota(j)}\}_{j\in[L]}$ instead of the commitment of $\{\mathsf{com}_j\}_{j\in[L]}$. In other words, our proposal is equivalent to the baseline ZKP, and corresponds with the baseline protocol but commiting $\{\mathsf{com}_{\iota(j)}\}_{j\in[L]}$ instead of $\{\mathsf{com}_j\}_{j\in[L]}$.

[7] The notation $\{0,1\}^{\leq h}$ refers to the set of all bit-strings with a length of at most $h \in \mathbb{N}$.

4.1 On the built-in algorithms of BAPC

At this point, we have all the ingredients to detail each built-in algorithm of the BAPC primitive. Our description focuses on the TCitH-based constructions in [4,9,34]; consequently, it allows an alternative instantiation of [13]. We present the two main built-in algorithms Build, Rebuild and Open of BAPC in Figures 4 to 6. We skip the presentation of the built-in algorithms Unfold, Commit (commit polynomials) and Verify (verify polynomial evaluations) of BAPC since they are the same as in the state-of-the-art.

Remark 4. We emphasize that the signatures in [4,13,9] employ $d = 2$, while [34] considers $d = 4$. That small change only affects the procedure to determine the value of $\mathbf{y}_{\mathbf{p}_\alpha}$ in Line 16 of the algorithm Open in Figure 6.

BAPC.Build($\mathbf{x}$)

1: Initialize an empty Stack to store nodes
2: Sample $\mathsf{z}_0 \leftarrow\$ \{0,1\}^\lambda$ and $\mathsf{salt} \leftarrow\$ \{0,1\}^{2\lambda}$
3: Committer.Init()
4: counter $\leftarrow 0$
5: Push the root $\mathbf{z}_\varnothing = \{\mathsf{z}_0, \varnothing\}$ onto Stack
6: Pop the first element in Stack, $\mathbf{z}_\mathsf{b} = \{\mathsf{z}_\mathsf{b}, \mathsf{b}\}$
7: Calculate $j = \iota^{-1}(\mathsf{b})$ such that $\mathsf{z}_\mathsf{b} = \mathsf{s}_j$
8: **if** $j < L - 1$ **then**
9: Get $(\mathsf{z}_{\mathsf{b}1}, \mathsf{z}_{\mathsf{b}0})$ from Expand(z_b; salt)
10: Push $\mathbf{z}_{\mathsf{b}1} = \{\mathsf{z}_{\mathsf{b}1}, \mathsf{b}1\}$ onto Stack
11: Push $\mathbf{z}_{\mathsf{b}0} = \{\mathsf{z}_{\mathsf{b}0}, \mathsf{b}0\}$ onto Stack
12: Go to Step 6
13: **if** $L - 1 \leq j < 2L - 1$ **then**
14: counter $\leftarrow$ counter $+ 1$
15: Compute $(e, i) = \psi^{-1}(j)$
16: Get $([\![\mathbf{x}^{(e)}]\!]_i, [\![\mathbf{u}^{(e)}]\!]_i)$ from PRG(z_b; salt)
17: $\Delta_\mathbf{x}^{(e)} \leftarrow \Delta_\mathbf{x}^{(e)} + [\![\mathbf{x}^{(e)}]\!]_i$
18: $\mathbf{p}_\mathbf{x}^{(e)} \leftarrow \mathbf{p}_\mathbf{x}^{(e)} + [\![\mathbf{x}^{(e)}]\!]_i \cdot (x + \phi_e(i))$
19: $\mathbf{p}_\mathbf{u}^{(e)} \leftarrow \mathbf{p}_\mathbf{u}^{(e)} + [\![\mathbf{u}^{(e)}]\!]_i \cdot (x + \phi_e(i))$
20: Determine $\mathsf{com}_i^{(e)}$ from Commit(z_b; salt)
21: Committer.Absorb($\mathsf{com}_i^{(e)}$)
22: Go to Step 6
23: **if** counter $= L$ **then**
24: $\Delta_\mathbf{x} \leftarrow \{\mathbf{x} - \Delta_\mathbf{x}^{(e)}\}_{e \in [\tau]}$
25: $\mathbf{p}_\mathbf{x} \leftarrow \{\mathbf{x} \cdot x^2 + \mathbf{p}_\mathbf{x}^{(e)}\}_{e \in [\tau]}$
26: com $\leftarrow$ Committer.Finalize()
27: decom $\leftarrow$ (salt, z_0)
28: **return** (com, decom, $\{\Delta_\mathbf{x}, \mathbf{p}_\mathbf{x}, \mathbf{p}_\mathbf{u}\}$)
29: **else**
30: Go to Step 6

Fig. 4: The Build algorithm: It employs the notation from Section 2, Definition 1, and Remark 3. In particular, $\mathbf{x}$ denotes a valid witness satisfying $f(\mathbf{x}) = 0$, for some public degree-2 polynomial system relation $f\colon \mathbb{F}^n \to \mathbb{F}^m$.

To highlight the main advantage of the unified primitive BAPC, we discuss the main difference between the baseline approach that stores all the $2L - 1$ tree nodes in the OBT and our algorithms in Figures 4 and 5. By design, algorithms in Figures 4 and 5 require a maximum of $\log_2(2L - 1)$ tree nodes. In addition, the commitments com, pcom and opcom along with the opening information $\mathsf{decom}_{I_\psi} \in \{0,1\}^{\mathsf{T}_\mathsf{open}\lambda}$ remain of the same size. However, we consider those strings as part of the signature output and, hence, no additional stack memory for storing those strings.

BAPC.Rebuild(opcom,decom$_{I_\psi}$,I,$\Delta_{\mathbf{x}}$)

1: Calculate the image $I_\psi = \psi(I)$
2: Calculate $\mathcal{J} =$ BAPC.Unfold(I_ψ)
3: Sort of indexes $\mathcal{J}$
4: Committer.Init()
5: counter $\leftarrow$ 0,k $\leftarrow$ 0, valid $\leftarrow$ $false$
6: Initialize an empty Stack to store nodes
7: Pop the first element in Stack, $\mathbf{z}_{\mathsf{b}} = \{\mathsf{z}_{\mathsf{b}}, \mathsf{b}\}$
8: $j \leftarrow \iota^{-1}(\mathsf{b})$
9: **if** $j = \mathcal{J}[\mathsf{k}]$ $\wedge$ valid **then**
10: Push decom$_{I_\psi}[\mathsf{k}]$ onto Stack
11: k $\leftarrow$ k + 1, valid $\leftarrow$ $true$
12: **if** $j < L - 1$ **then**
13: **if** valid $= true$ **then**
14: Get $(\mathsf{z}_{\mathsf{b}1}, \mathsf{z}_{\mathsf{b}0})$ from Expand(z_{b}; salt)
15: Push $\mathbf{z}_{\mathsf{b}1} = \{\mathsf{z}_{\mathsf{b}1}, \mathsf{b}1\}$ onto Stack
16: Push $\mathbf{z}_{\mathsf{b}0} = \{\mathsf{z}_{\mathsf{b}0}, \mathsf{b}0\}$ onto Stack
17: Go to Step 7
18: **if** $L - 1 \leq j < 2L - 1$ **then**
19: counter $\leftarrow$ counter + 1
20: **if** valid $= true$ **then**
21: $(e, i) \leftarrow \psi^{-1}(j)$
22: pt $\leftarrow \phi_e(I_\psi^{(e)}) - \phi_e(i)$
23: Get $([\![\mathbf{x}^{(e)}]\!]_i, [\![\mathbf{u}^{(e)}]\!]_i)$ from PRG(z_{b}; salt)
24: $\mathbf{y}_{\mathbf{p_x}}^{(e)} \leftarrow \mathbf{y}_{\mathbf{p_x}}^{(e)} + [\![\mathbf{x}^{(e)}]\!]_i \cdot$ pt
25: $\mathbf{y}_{\mathbf{p_u}}^{(e)} \leftarrow \mathbf{y}_{\mathbf{p_u}}^{(e)} + [\![\mathbf{x}^{(e)}]\!]_i \cdot$ pt
26: Determine com$_i^{(e)}$ from Commit(z_{b}; salt)
27: Committer.Absorb(com$_i^{(e)}$)
28: **else**
29: Committer.Absorb(opcom$_i^{(e)}$)
30: Go to Step 7
31: **if** counter $= L$ **then**
32: $\mathbf{y}_{\mathbf{p_x}} \leftarrow \{\mathbf{y}_{\mathbf{p_x}}^{(e)} - \phi_e(I_\psi^{(e)}) \cdot \Delta_{\mathbf{x}}^{(e)}\}_{e \in [\tau]}$
33: $\mathbf{y}_{\mathbf{p_u}} \leftarrow \{\mathbf{y}_{\mathbf{p_u}}^{(e)} - \phi_e(I_\psi^{(e)}) \cdot \Delta_{\mathbf{x}}^{(e)}\}_{e \in [\tau]}$
34: com $\leftarrow$ Committer.Finalize()
35: **return** $(\mathsf{com}, \{\mathbf{y}_{\mathbf{p_x}}, \mathbf{y}_{\mathbf{p_u}}\})$
36: **else**
37: Go to Step 7

Fig. 5: The Rebuild algorithm: Same description as in Figure 4.

Stack Memory Considerations for the Prover: Notice that the Prover in Figure 3 requires the length-τ list of polynomials $\mathbf{p_x}, \mathbf{p_u}$ along with the correction values $\Delta_{\mathbf{x}}$. In the baseline approach, all these polynomials and correction values are stored, whereas our memory-less approach proceeds as follows. Clearly, one could compute $\mathbf{p}_{\mathbf{x}}^{(e)}, \mathbf{p}_{\mathbf{u}}^{(e)}$ and $\Delta_{\mathbf{x}}^{(e)}$ on the fly; however, these polynomials are required in the computation of $\mathbf{p}_\alpha$ as determined by the challenge $\mathbf{\Gamma}$ (usually generated from the commitment of the seeds), which implies a considerable overhead in the calculations, as we would be forced to traverse OBT τ more times. Hence, we consider the scenario where we store these τ polynomials and correction values. Since the polynomials $\mathbf{p_u}$ and $\mathbf{p}_\alpha$ belong to the same space, and $\mathbf{p_u}$ is only used for the calculation of $\mathbf{p}_\alpha$, we assume that the polynomial $\mathbf{p_u}$ is stored at the same memory address as $\mathbf{p}_\alpha$ since $\deg(\mathbf{p_u}) \leq \deg(\mathbf{p}_\alpha)$, without increasing the use of stack memory. On the other hand, the polynomial data evaluation $\mathbf{y}_{\mathbf{p}_\alpha}$ in the algorithm's output in Figure 6 is part of the signature; hence, we also assume that there is no additional memory requirement for $\mathbf{y}_{\mathbf{p}_\alpha}$.

Stack Memory Considerations for the Verifier: Observe that the Verifier in Figure 3 requires the length-τ list of polynomial evaluations $\mathbf{y}_{\mathbf{p_x}}, \mathbf{y}_{\mathbf{p_u}}$ along with the correction values $\Delta_{\mathbf{x}}$. In the baseline approach, all these polynomial evaluations and correction values are stored. In contrast, our memory-less approach proceeds as follows. We find that the polynomial evaluations $\mathbf{y}_{\mathbf{p_u}}$ and $\mathbf{y}_{\mathbf{p}_\alpha}$ belong to the same space and that $\mathbf{y}_{\mathbf{p_u}}$ is used only to compute the polynomial $\mathbf{p}_\alpha$. Therefore,

BAPC.Open(decom,idcom$_{I_\psi}$,I,$\mathbf{p}_\alpha$)

Compute decom_{I_ψ} from idcom_{I_ψ}
for each $e \in [\tau]_0$ **do**
  $\iota_j \leftarrow \iota(L-1+\psi(i_e))$
  Calculate the tree height of the $(L-1+\psi(i_e))$-th leaf from ι_j
  $\mathsf{bits} \leftarrow$ bit representation of ι_j
  Obtain z_0 and salt from decom
  Determine $(\mathsf{z}_1, \mathsf{z}_0)$ from $\mathsf{Expand}(\mathsf{z}_0; \mathsf{salt})$
  for each $i \in [\mathsf{height}]_0$ **do**
    Compute the index $\mathsf{current}_{\iota_j}$ at level $(\mathsf{height}-i-1)$ determined by ι_j
    $\mathsf{current}_j \leftarrow \iota^{-1}(\mathsf{current}_{\iota_j})$
    $\mathsf{z}_0 \leftarrow \mathsf{z}_{\mathsf{bits}_0 \oplus 1}$
    Remove bits_0 from bits
    Determine $(\mathsf{z}_1, \mathsf{z}_0)$ from $\mathsf{Expand}(\mathsf{z}_0; \mathsf{salt})$ and the index $\mathsf{current}_j$
  $\mathsf{z}_0 \leftarrow \mathsf{z}_{\mathsf{bits}_0 \oplus 1}$
  Calculate opcom_e from $\mathsf{Commit}(\mathsf{z}_0; \mathsf{salt})$ and the index $\psi(i_e)$
  Determine $\mathbf{y}^{(e)}_{\mathbf{p}_\alpha}$ from $\mathbf{p}^{(e)}_\alpha$ and $\mathsf{current}_j$
return $(\mathsf{opcom}, \mathbf{y}_{\mathbf{p}_\alpha}, \mathsf{decom}_{I_\psi})$

Fig. 6: The Open algorithm: Same description as in Figure 4.

we assume that no additional memory is required for the constant coefficient of $\mathbf{y}_{\mathbf{p_u}}$ and $\mathbf{y}_{\mathbf{p}_\alpha}$, as they can be stored at the same memory address as $\mathbf{p}_\alpha$ (similarly to the Prover-side considerations).

Given the above stack memory considerations, Table 1 presents concrete numbers for the TCitH-based signatures in [4,34,13,9] under our framework modeled in Figure 3. We report the stack memory for OBT in both the baseline approach and our proposal. We noted that the stack memory bottleneck is now determined by the vector polynomial $\mathbf{p}_\mathbf{x}$. Nevertheless, as mentioned previously, one way to further reduce stack memory would be to compute both $\mathbf{p}^{(e)}_\mathbf{x}$ on the fly, at the cost of considerable performance overhead. Remarkably, fast variants of the TCitH-based signatures in [4,13,9,34] are expected to require a larger stack memory (at least in our approach) as they employ larger value of τ (this is also validated in our experiments in Section 6). Table 1 highlights the importance of small witnesses, since $\mathbf{p}_\mathbf{x}$ shares around the same dimensions as for $\mathbf{x}$ but is defined over a field extension. For example, in SDitH $\mathbf{x} \in \mathbb{F}_2^n$ with $n \in \{10360, 18396, 19864\}$, while $\mathbf{p}_\mathbf{x} \in (\mathbb{K}^n)^\tau$ where $\mathbb{K} = \mathbb{F}_{2^\kappa}$ with $\kappa \in \{8, 11\}$, which leads to a significant memory requirement. In contrast, MQOM employs $\mathbf{x} \in \mathbb{F}_{16}^n$ with $\{56, 84, 116\}$, which produces the smallest $\mathbf{p}_\mathbf{x}$ sizes among the schemes in Table 1. Finally, we emphasize that our proposal ensures that no additional re-computations are required for the tree and polynomials. However, if performance is not a priority, one can avoid storing $\mathbf{p}_\mathbf{x}$ by allowing recomputation of the GGM tree (we discuss this in more detail in Section 7).

Scheme	σ	OBT		Δ_x	p_x	p_u	Overall
		Baseline	This work				
Mirath-1s	3.2	786.4	**0.3**	0.7	8.1	0.2	12.4
Mirath-1f	3.7	139.2	**0.2**	1.0	7.6	0.3	12.8
Mirath-3s	7.5	1867.8	**0.4**	1.6	18.8	0.5	28.7
Mirath-3f	8.5	331.8	**0.3**	2.2	17.8	0.6	29.6
Mirath-5s	13.1	3276.8	**0.5**	2.9	34.2	0.8	51.5
Mirath-5f	15.4	589.8	**0.5**	4.1	32.8	1.2	54.0
MQOM-1s	2.9	786.4	**0.3**	0.3	1.5	0.2	5.2
MQOM-1f	3.3	139.2	**0.2**	0.5	1.2	0.3	5.5
MQOM-3s	6.5	1769.4	**0.4**	0.8	3.5	0.4	11.5
MQOM-3f	7.7	331.8	**0.3**	1.1	2.9	0.6	12.8
MQOM-5s	12.0	3276.8	**0.5**	1.4	6.6	0.8	21.4
MQOM-5f	13.8	589.8	**0.5**	1.5	3.8	0.8	20.4
RYDE-1s	3.1	786.4	**0.3**	0.7	19.1	0.3	23.4
RYDE-1f	3.6	139.2	**0.2**	1.0	27.1	0.4	32.2
RYDE-3s	7.1	1867.8	**0.4**	1.4	43.2	0.6	52.6
RYDE-3f	8.3	331.8	**0.3**	1.9	61.3	0.9	72.8
RYDE-5s	12.6	3276.8	**0.5**	2.6	71.2	1	88.0
RYDE-5f	14.8	589.8	**0.5**	3.7	102.5	1.4	123.0
SDitH-1s	4.3	917.5	**0.3**	18.1	202.0	2.5	227.2
SDitH-1f	5.5	172.0	**0.2**	27.2	220.2	2.7	255.9
SDitH-3s	8.4	1867.8	**0.4**	43.7	485.8	5.2	543.5
SDitH-3f	11.4	380.9	**0.3**	71.3	576.2	6.0	665.2
SDitH-5s	15.6	3276.8	**0.5**	62.1	691.9	9.1	779.2
SDitH-5f	20.0	688.1	**0.5**	104.3	845.0	10.8	980.5

Table 1: Stack memory: All values are given in **KB**. We report for the fast and short variant of each scheme. We consider the fastest variants of Mirath and MQOM with $\mathbb{F} = \mathbb{F}_{16}$. In addition, we consider the five-round variant of MQOM, as it can be modeled as in Figure 3. We also report numbers for the TCitH variant of SDitH using the underlying-problem and proof-system parameters from [34, Tables 3 and 9].

4.2 Comparison with the State-of-the-Art

Notice that the idea of performing a Depth-First-Search (DFS) instead of a Breadth-First-Search (BFS) to traverse a tree is not new. For instance, [10] describes and implements this idea at a high level for the forest approach, leaving the implementation of the OBT scenario as future work. The recent work in [14] provides an implementation of the DFS approach for the OBT scenario, independently of our work. The MQOM design is based on the secret-key root correlated GGM tree technique from [26,29], and its security is considered heuristic in the Random Oracle Model (ROM) according to [31]. Although, these findings do not lead to an attack on MQOM and recent results suggest a way to fix the issue [32,24]. We have conservatively excluded the use of the secret-key root part of the optimization and instead use only the half tree technique, but our results

should easily apply to the correlated half trees as in [13]. We provide a more detailed comparison in Section 6.2.

In summary, we provide the permutation that links DFS and BFS to traverse OBT. We leverage this equivalence to provide a simple and efficient implementation for processing and committing each leaf (seed) in the OBT, along with the calculation of the required polynomials and their evaluations, including the opening of the commitments for the challenges determined by the index list L. Our framework in Figure 3 can be implemented without additional recomputation of OBT and polynomials; making it as efficient as possible. Furthermore, we believe that our framework and implementation can help the cryptographic community develop new TCitH-based signature schemes taking into account memory bottlenecks for constrained devices. Our implementation can serve as a skeleton for implementing new signature schemes, as it isolates the tasks of implementing the underlying arithmetic and the polynomial proof procedures.

5 AES and Rijndael-256 Implementations

MPCitH schemes submitted to the NIST on-ramp competition usually instantiate GGM trees using AES-128 for the $\lambda = 128$ security level and Rijndael-256 for $\lambda = 192$ and 256, or Keccak-based primitives (SHA3/SHAKE) for all levels. During the AES standardization process, Rijndael was proposed as a parameterized block cipher supporting multiple block and key sizes; however, NIST only standardized a 128-bit block size and 128-, 192- and 256-bit key sizes as AES. Recently [35], NIST expressed interest in standardizing Rijndael with 256-bit block and key sizes, which we refer to as Rijndael-256 in the rest of this section.

As our memory-less algorithms are particularly beneficial for embedded architectures, we showcase their application on the ARM Cortex-M4 core, based on the ARMv7-M architecture; it is the most popular choice for embedded implementations of schemes in NIST's on-ramp competition and the previous NIST PQC competition. The pqm4 project [27] collects many such implementations.

However, securely implementing AES/Rijndael in the Cortex-M4 is a non-trivial task. When Rijndael was proposed, efficient software implementations relied on lookup tables indexed by secret data, which were later shown to be susceptible to cache timing attacks [19,3,18]. It is possible, in principle, to realize constant-time memory accesses in many commercial Cortex-M4 parts, but doing so requires careful analysis of target device characteristics: flash memory is often cached (e.g. the ART Accelerator in ST Microeletronics STM32 parts); some parts include SRAM caches, such as NXP i.MX RT1170 and i.MX 7; even if SRAM caches are not used, MCUs with a banked RAM organization may display timing differences when accessing neigboring memory address at bank boundaries, a phenomenon observed in [36] and independently replicated in [38, Section 4.1]. Code written to target a wide variety of Cortex-M4 parts may find it difficult, if not impossible, to guarantee constant-time memory accesses. In light of the risk of timing attacks, it might be argued that the safer path is to eschew the use

of lookup tables in favor of fully constant-time AES/Rijndael implementations, although this has non-trivial performance implications.

There is much prior work on efficient and constant-time Cortex-M4 implementations of Keccak, AES-128 and AES-256 (AES using 128-bit blocks and 256-bit keys) [40,7,5,25,2], usually with accompanying freely available source code; for these primitives, we reuse these state of the art implementations. However, interest in Rijndael-256 has only picked up recently; a literature search only uncovered works that re-purpose AES acceleration instructions for the Intel architecture [23] and RISC-V [39] to implement Rijndael-256. They do not apply to the Cortex-M4, which lacks AES acceleration instructions[8]. A portable 64-bit implementation of Rijndael-256 is used in [14], obtained from the BearSSL library [37], but the authors recognize that there is space to improve its performance for the Cortex-M4. In light of this gap, we decided to write a novel, high-performance, constant-time Rijndael-256 implementation optimized for 32-bit devices (and to some degree the Cortex-M4 core), which we describe later.

5.1 A review of AES and existing constant-time implementations

We first review AES and the main techniques in the implementations of Schwabe and Stoffelen [40], which serves as the basis of our Rijndael-256 implementation, and Adomnicai and Peyrin [7] as the state of the art in Cortex-M4 performance.

The AES state consists of 16 bytes (elements of $\mathbb{F}_{2^8}$ modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$), laid out in a 4×4 state matrix and linearized in column-major order, packing together bits of each $\mathbb{F}_{2^8}$ element in a byte using a polynomial-basis representation; for more details, we refer to the AES specification [8] and the book by Daemen and Rijmen [22]. An AES round is broken down into four steps: SubBytes, ShiftRows, MixColumns and AddRoundKey, of which SubBytes is most difficult to implement in constant-time; it consists of the parallel application to each byte of an S-box, defined as a field inversion in $\mathbb{F}_{2^8}$ followed by an affine transformation. If lookup tables cannot be used, due to cache-timing attacks, an alternative is to interpret it as a Boolean function mapping 8 bits of input to 8 bits of output, for which equivalent logic circuits are known, such as a 113-gate one proposed by Boyar, Matthews and Peralta [20].

To take advantage of this observation in software implementations, one must transpose the conventional representation: rather than packing bits of an element of $\mathbb{F}_{2^8}$ together in the same byte, they are spread out across different machine words. This allows evaluating the S-box's equivalent logical circuit, using bitwise logical instructions, in parallel for each byte of the state matrix. This technique is known as *bitslicing* and was pioneered by Biham [17] for DES. It was applied to AES on Intel (prior to the introduction of AES acceleration instructions) by Mitsui and Nakajima [33]. Further performance improvements were also demonstrated in the Intel architecture by Käsper and Schwabe [28].

[8] Although some Cortex-M4 MCUs feature vendor-specific AES acceleration coprocessors, these are often not flexible enough to implement Rijndael-256.

Schwabe and Stoffelen [40] optimize AES for the Cortex-M4 core, exploiting ARMv7-M instructions for bitfield extraction and logical operations with shifted/rotated operands, and an optimized register allocation for the Boyer-Matthews-Peralta S-box circuit. For ARMv7-M, a 32-bit architecture, an 8-word representation comprises a total of 256 bits, whereas the AES state only requires 128 bits. Schwabe and Stoffelen pack two independent AES states that can be operated in parallel, i.e. key scheduling and encryption of two independent keys and plaintexts. This proves useful for modes of operation with inherent parallelism such as CTR mode, as well as its application to GGM trees.

Although the canonical AES representation employs a column-major linearization of the state matrix, Käsper and Schwabe argue that using row-major simplifies the MixColumns step of AES. This choice is replicated in the Cortex-M4 implementations of Schwabe and Stoffelen, and Adomnicai and Peyrin, and is displayed in Figure 7. Note that b_i^j $(0 \leq i < 128,\ j \in \{0,1\})$ denotes the i-th bit of the conventional AES representation for the j-th independent state.

	row 3								·····	row 0							
	column 0		column 1		column 2		column 3		·····	column 0		column 1		column 2		column 3	
	block 0	block 1	block 0	block 1	block 0	block 1	block 0	block 1	·····	block 0	block 1	block 0	block 1	block 0	block 1	block 0	block 1
R_0	b_{24}^0	b_{24}^1	b_{56}^0	b_{56}^1	b_{88}^0	b_{88}^1	b_{120}^0	b_{120}^1	·····	b_0^0	b_0^1	b_{32}^0	b_{32}^1	b_{64}^0	b_{64}^1	b_{96}^0	b_{96}^1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	·····	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
R_7	b_{31}^0	b_{31}^1	b_{63}^0	b_{63}^1	b_{95}^0	b_{95}^1	b_{127}^0	b_{127}^1	·····	b_7^0	b_7^1	b_{39}^0	b_{39}^1	b_{71}^0	b_{71}^1	b_{103}^0	b_{103}^1

Fig. 7: Representation of two independent AES states in the implementations of Schwabe and Stoffelen [40] and Adomnicai and Peyrin [7]. Reproduced from [7].

Schwabe and Stoffelen implement ShiftRows using `ubfx` (unsigned bitfield extract) and XOR instructions, while MixColumns benefits from ARMv7-M's logical instructions with rotated operands. AddRoundKey, as in the conventional representation, is a sequence of XOR instructions, so long as the round keys also employ the same bitsliced representation as the state. This is achieved by implementing the key schedule in a bitsliced fashion, which also ensures constant-time evaluation of the S-box. For interoperability, AES plaintexts (resp. ciphertexts) must be input (resp. output) in conventional representation. Schwabe and Stoffelen show an efficient procedure to convert from/to bitsliced representation using 12 SWAPMOVE operations, which swaps bits (selected by a bitwise mask) between two words; it is also applied to the master key during key scheduling.

Adomnicai and Peyrin improve upon Schwabe and Stoffelen's implementation by the application of a technique called fixslicing, originally proposed for the GIFT cipher by Adomnicai, Najm and Peyrin [6]. The main insight is to avoid explicitly performing the ShiftRows step, fixing the per-row permutation of the elements (hence the name). Consequently, the other AES steps need to

be modified to take into account the new positions of each element, which are different from round to round. We refrain from discussing such modifications, as we opted not to apply fixslicing to our Rijndael-256 implementation.

5.2 Our new constant-time Rijndael-256 implementation

Our implementation roughly follows that of Schwabe and Stoffelen for AES. A major difference between AES and Rijndael-256 is the use of 256-bit block and key sizes, which matches the state size for 32-bit architectures when using an 8-word representation, eschewing the need to pack two independent states.

We adopt a row-major linearization of the state matrix, as in previous AES implementations. Our chosen representation maps the first row of the AES state to bits $0, \ldots, 7$, the second row to bits $8, \ldots, 15$, and so on, as shown in Figure 8. The indexing of each bit b_i is defined similarly as for Figure 7, except that we drop the superscript index identifying independent AES states.

	row 3								$\cdots$	row 0							
	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7	$\cdots$	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7
R_0	b_{24}	b_{56}	b_{88}	b_{120}	b_{152}	b_{184}	b_{216}	b_{248}	$\cdots$	b_0	b_{32}	b_{64}	b_{96}	b_{128}	b_{160}	b_{192}	b_{224}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
R_7	b_{31}	b_{63}	b_{95}	b_{127}	b_{159}	b_{191}	b_{223}	b_{255}	$\cdots$	b_7	b_{39}	b_{71}	b_{103}	b_{135}	b_{167}	b_{199}	b_{231}

Fig. 8: Representation of a Rijndael-256 state in our implementation.

Since fixslicing is the state of the art in Cortex-M4 AES performance, we considered its use for Rijndael-256. However, the lack of regularity in Rijndael-256's ShiftRows offsets, versus those of AES, presents a problem; the following explanation is easier to follow by referring to [7, Figures 6 and 7]. The computation of MixColumns in Adomnicai and Peyrin's implementation calls for both 32-bit rotations and bytewise rotations applied to each byte of a 32-bit word, in SIMD fashion, using identical rotation amounts for all bytes of the word; recall that each byte maps to a column of the AES state as shown in Figure 7.

In architectures without SIMD shift/rotation instructions, such as ARMv7-M, the efficiency of the fixslicing technique implicitly relies on using the same rotation amounts for each byte, which allows bytewise rotations to be implemented with only two 32-bit shifts (one to the left and one to the right), plus other operations, as in the following C preprocessor macro used in Adomnicai and Peyrin's implementation for bytewise right-rotation of a 32-bit word by 2:

```
#define BYTE_ROR_2(x) ((((x) >> 2) & 0x3f3f3f3f) | (((x) & 0x03030303) << 6))
```

We represent the AES ShiftRows offsets as $[0, 1, 2, 3]$, meaning that the first row of the AES state matrix is not rotated, the second row is left-rotated by one element, and so on. Note the regular spacing of these offsets. which is not the case for those of Rijndael-256 ($[0, 1, 3, 4]$). The resulting bit permutations in the Rijndael-256 state require performing bytewise rotations with distinct per-byte

Listing 1.1: Naïve implementation of ShiftRows for Rijndael-256.

```
1  // Bytewise rotation of second, third and fourth bytes
2  uint32_t s1 = ((s & 0x00007F00) << 1) | ((s & 0x00008000) >> 7);
3  uint32_t s2 = ((s & 0x001F0000) << 3) | ((s & 0x00E00000) >> 5);
4  uint32_t s3 = ((s & 0x0F000000) << 4) | ((s & 0xF0000000) >> 4);
5  // Combine all bytes
6  s = (s & 0x000000FF) | s1 | s2 | s3;
```

rotation amounts, which are more expensive: a 32-bit word must be split into 4 bytes, each byte must be separately rotated, and these bytes must be combined to produce the output 32-bit word. This is shown in C code form in Listing 1.1 discussed next. We estimated operation counts and concluded that a fixsliced Rijndael-256 implementation is unlikely to outperform one that uses ShiftRows.

In light of this, we opt for an explicit ShiftRows step, as in Schwabe and Stoffelen's AES implementation. Due to the similar (row-major) representation, SubBytes, MixColumns and AddRoundKey can be implemented similarly as in AES, but the distinct offsets in Rijndael-256 require a new version of ShiftRows. While a similar strategy as in Schwabe and Stoffelen's implementation could be used, we propose a new trick to implement ShiftRows with fewer instructions, which applies to both AES and Rijndael-256.

In the bitsliced row-major representation, ShiftRows is seen as a rotation of each byte of the 8-word state by a distinct amount: from least- to most-significant byte, each is rotated by 0, 1, 3 and 4 bits. Concretely, the C code in Listing 1.1 is a naïve implementation of ShiftRows, applied in-place to a single word `s`.

This appears to require 19 CPU instructions: 7 ANDs, 6 shifts and 6 ORs. However, ARMv7-M allows merging shifts with logical operations in a single instruction. The resulting count of 13 instructions matches Schwabe and Stoffelen's `ubfx` approach, except that some bitmasks must be loaded to registers (a cost which can be amortized across 8 ShiftRows applications, one for each word), whereas for `ubfx`, constants are directly encoded inside the instruction.

We propose an improvement by observing that rotations can be composed, with the effect of adding rotation amounts. We rewrite Rijndael-256's ShiftRows offsets as $[0, 1, 3, 4] = [0, 1, 0 + 3, 1 + 3] = [0, 1, 0, 1] \circ [0, 0, 3, 3]$; in both steps, non-zero rotation amounts per byte are identical. Using a SIMD-within-a-register technique, a single operation implements these identical rotations across different bytes (plus some additional operations to combine with non-rotated bytes). Thus, only two SIMD-within-a-register bytewise rotations with bit masking need to be implemented, saving one bytewise rotation compared to the naïve version.

Listing 1.2 in Appendix A implements this idea, again applying ShiftRows in-place to a single word `s`. It appears to require 6 ANDs, 4 shifts and 4 ORs, but shifts can be merged with other operations to yield only 10 CPU instructions. Moreover, `t |= s & 0x0000FFFF` can be replaced by a `bfi` (bitfield insert) instruction for a final tally of 9 instructions. In principle, 5 bitmasks must be loaded

beforehand (a cost amortized across applications of ShiftRows to all 8 words). By some transformations and careful coding, this can be reduced to explicit loading of only 2 bitmasks, while the other 3 are encoded as modified immediates of AND instructions, without increasing instruction count. This is demonstrated in the assembly language implementation of Listing 1.3 in Appendix A.

Note that an analogous trick works for the AES ShiftRows offsets, by composing rotations as $[0,1,2,3] = [0,1,0+2,1+2] = [0,1,0,1] \circ [0,0,2,2]$, and results in the same instruction count. It also works if two AES states are packed into 8 words, in which case the bytewise rotations are by 0, 2, 4 and 6 bits, which can be decomposed as 0, 2, $4+0$ and $4+2$.

6 Experimental Results

We now display performance, code size and stack usage for signature schemes implemented using our framework, as well as Rijndael-256 microbenchmarks.

We consider TCitH-based signatures with publicly available implementations and plug them into our ZKP framework in Figure 3. We do not integrate SDitH as there is no implementation of the TCitH variant; implementing it from scratch would require considerable effort beyond the scope of the paper. On the other hand, we omit implementing MQOM within our framework and comparing against the recent MQOM implementation results for Cortex-M4 in [14]. Hence, we showcase the feasibility and memory savings potential of our proposed techniques by benchmarking the schemes Mirath and RYDE on both the Intel 64-bit architecture and the Cortex-M4. Yet, we discuss in Section 7 about potential improvements of MQOM and SDitH under our ZKP framework.

The authors of the Mirath and RYDE schemes state that their implementations are intended to be constant-time. In particular, RYDE explicitly aims to provide constant-time implementation for the reference code, while for Mirath this property is not explicitly guaranteed for the reference implementation. Since our framework itself does not introduce secret-dependent branches or memory accesses, and it only orchestrates calls to the underlying arithmetic routines, the constant-time properties of the resulting implementation depend primarily on the guarantees provided by the underlying scheme implementations.

The implementations whose results are reported in this section are made available with a permissible open source license at [21].

6.1 Benchmarking methodology

Host-side benchmarks are run on a machine with an Intel Core i7-13700K CPU with nominal core clock of 3.4 GHz for its performance cores, running Ubuntu Linux 25.10. The compiler used is clang 20.1.8. Hyperthreading is turned off at the computer setup level. Although we measure clock cycles, which are to a first order immune to clock frequency fluctuations, care is taken to avoid such fluctuations: TurboBoost is disabled to prevent upclocking, and the Linux scaling

governor `performance` is used to avoid downclocking. The CPU has an adequate cooling solution and the machine is kept in a temperature controlled room to prevent thermally induced clock throttling. Lastly, benchmark execution is pinned to a specific CPU core to reduce the effect of context switches.

Performance figures on Intel are obtained as an average of 100 runs. We measure cycle counts using the `libcpucycles` library by D. J. Bernstein [15], which accesses the CPU's performance counters.

Our Cortex-M4 test platform is the ST Microelectronics NUCLEO-L4R5ZI board, based on the STM32L4R5ZI MCU, which includes 2 MB of flash memory and 640 KB of RAM. We have ported Mirath and RYDE to pqm4 [27], whose benchmarking scripts configure the microcontroller to run at a fixed 20 MHz clock frequency, so as to avoid flash wait states that introduce unpredictability and slow down code execution. The cross-compiler used is arm-none-eabi-gcc version 14.2.1, and we run the pqm4 benchmark script with the `-o speed` option to optimize for speed, which employs the gcc `-O3` option.

Note that the NIST round-2 submissions of Mirath and RYDE use too much RAM to run on commercially available microcontrollers, and even for the QEMU emulator as employed by pqm4 for host-side testing of Cortex-M4 binaries. We have modified the linker scripts included in pqm4 for use with QEMU to increase the available RAM from 4 MB to 16 MB, which is sufficient to run all parameter sets of both schemes in their original form. As will be seen, the use of our memory-less algorithms allows all parameter sets to run directly on-device, despite the limited amount of RAM available.

6.2 Experiments and Comparisons

In Table 2, we provide Cortex-M4 microbenchmarks for Rijndael-256. For comparison purposes, we also display results for AES-128, using a slightly modified version of the assembly language implementation of [7]; we recall that this implementation performs two key schedules/encryptions in parallel, so the amortized cost is cut in half if parallel operations are performed. Cycle counts for key scheduling include the cost of converting the master key to bitsliced representation, and for encryption, the cost of converting the plaintext to bitsliced representation and the ciphertext to conventional representation. Our Rijndael-256 implementation is 2.66× faster than that of [14], and our constant-time key schedule implementation is only 4.2% slower than the LUT-based version of [14].

In Table 3, we present code size and stack usage figures in the Cortex-M4 for Mirath and RYDE, both ported directly from their round-2 NIST submissions as well as using our implementation. We also report memory savings by the use of our techniques, in comparison to the baseline implementation of the same variant, as the ratio $1 - s_{Tw}/s_B$, where s_B and s_{Tw} denote the stack usage for the baseline implementation and this work, respectively.

These figures are obtained by running the pqm4 `benchmarks.py` script using the QEMU emulator on the host, which is required to present figures for most of the parameter sets, which do not fit on the STM32L4R5ZI. Code size figures are a static property of the compiled binaries, and to a first order not affected

Cipher	Operation	Implementation	Cycle count	LUT-based
AES-128	Key schedule	[7]	4279	No
AES-128	Encryption	[7]	3007	No
Rijndael-256	Key schedule	[14]	10932	Yes
Rijndael-256	Encryption	[14]	16160	No
Rijndael-256	Key schedule	This work	11396	No
Rijndael-256	Encryption	This work	6069	No

Table 2: Cortex-M4 performance of AES and Rijndael-256 implementations.

by the platform the code runs on (although there are small differences due to hardware setup code). Stack usage should be identical regardless of whether the code is run on an emulator or the actual device, but again, most parameter sets cannot fit on the hardware without our memory-less algorithms.

The main takeaway from Table 3 is that our approach immensely reduces the stack usage compared to baseline implementations, by up to 99%. This does not come at a significant increase in code size; it is even slightly reduced in some cases. Remarkably, our experiments show a smaller stack memory requirement compared to FAEST. Comparing with the theoretical stack memory numbers of Table 1, we note that Table 1 provides smaller numbers than the measured and presented in Table 3. Most of this difference comes from the stack memory requirement for challenge $\mathbf{\Gamma}$ in the implementations of Mirath and RYDE (they store the full matrix). For example, Mirath-5s requires a 24-by-229 matrix defined over $\mathbb{F}_{16^3}$, yielding a total of 7.3 KB. The additional measured stack memory comes from matrix $\mathbf{\Gamma}$ for each experiment of Mirath and RYDE. However, one can avoid storing the matrix $\mathbf{\Gamma}$ by processing each column/row in a streamlined fashion, thus reducing stack memory requirements for $\mathbf{\Gamma}$ considerably.

We also report on performance figures, both for the Intel architecture and the Cortex-M4, in Table 3. Baseline implementations which exceed the available RAM (640 KB) in the STM32L4R5ZI microcontroller are marked with "–". The figures seem to suggest some level of performance regressions due to the use of memory-less techniques, especially for Mirath. However, our memory-less impleme ntation does not take advantage of parallel evaluation of Keccak (i.e. Keccak $\times 4$), which accounts for some of the performance gap. Indeed, for RYDE we made slight modifications to the build system of its round-2 NIST submission package, so as to disable the use of Keccak $\times 4$, and it is clearly seen that the gap between baseline and memory-less implementations is significantly reduced. This suggests an important avenue of future implementation work, which is to integrate Keccak $\times 4$ with our memory-less techniques.

7 Additional Discussion and Concluding Remarks

Our unified TCitH framework enables us to identify the new stack memory bottleneck involving OBT and polynomials. Our algorithms enable efficient implementations that are memory-friendly to resource-constrained devices such as

Scheme	Impl.	Code size (KB)	Stack usage (KB)			Performance (Mcycles)					
						Intel (AVX)			Cortex-M4		
			Keygen	Sign	Verify	Keygen	Sign	Verify	Keygen	Sign	Verify
Mirath-1s	Baseline	38.2	8.78	1204	1228	0.071	15.9	13.0	–	–	–
Mirath-1f	Baseline	41.7	8.78	232	236	0.072	4.88	2.60	1.25	318	238
Mirath-3s	Baseline	75.0	17.1	2852	2890	0.174	84.0	79.6	–	–	–
Mirath-3f	Baseline	80.3	17.1	546	551	0.174	20.6	14.9	2.69	1020	939
Mirath-5s	Baseline	73.9	30.4	5003	5052	0.295	111	97.8	–	–	–
Mirath-5f	Baseline	71.2	30.4	970	976	0.298	34.4	26.1	–	–	–
Mirath-1s	**This work**	38.3	8.87	19.0 (98%)	19.8 (98%)	0.064	20.9	17.1	1.24	1597	1479
Mirath-1f	**This work**	42.7	8.87	16.8 (93%)	18.6 (92%)	0.066	6.80	3.35	1.24	286	228
Mirath-3s	**This work**	71.8	17.1	37.3 (99%)	38.4 (99%)	0.158	93.2	91.1	2.68	5822	5651
Mirath-3f	**This work**	77.4	17.1	34.2 (94%)	35.9 (93%)	0.164	21.8	16.3	2.68	1135	1014
Mirath-5s	**This work**	70.5	30.4	64.4 (99%)	65.1 (99%)	0.262	139	124	4.87	8696	8246
Mirath-5f	**This work**	69.2	30.4	60.4 (94%)	62.6 (94%)	0.263	37.3	30.0	4.87	2101	1921
RYDE-1s	Baseline	25.3	6.00	1246	1269	0.036	22.0	19.3	–	–	–
RYDE-1f	Baseline	25.5	6.00	294	296	0.036	8.28	3.48	1.32	371	321
RYDE-3s	Baseline	62.4	7.59	2938	2976	0.057	89.7	78.8	–	–	–
RYDE-3f	Baseline	62.8	7.59	676	682	0.056	18.5	14.4	–	–	–
RYDE-5s	Baseline	62.6	8.61	5142	5192	0.061	150	139	–	–	–
RYDE-5f	Baseline	62.9	8.61	1189	1195	0.062	30.6	25.6	–	–	–
RYDE-1s	**This work**	27.3	6.17	34.6 (97%)	36.5 (97%)	0.030	23.9	20.8	1.26	1713	1658
RYDE-1f	**This work**	28.5	6.17	43.7 (85%)	45.7 (85%)	0.030	8.34	3.79	1.26	394	318
RYDE-3s	**This work**	60.3	7.90	67.8 (98%)	70.0 (98%)	0.048	88.6	81.2	2.03	5902	5805
RYDE-3f	**This work**	60.3	7.90	88.6 (87%)	91.0 (87%)	0.048	17.4	14.7	2.03	1196	1090
RYDE-5s	**This work**	59.9	8.93	108 (98%)	111 (98%)	0.054	146	142	2.28	8806	8614
RYDE-5f	**This work**	59.8	8.93	145 (88%)	149 (88%)	0.054	30.3	26.0	2.28	1823	1646
FAEST-1s	[10]	–	0.81	31.3	31.2	–	–	–	0.080	1288	1256
FAEST-1f	[10]	–	0.81	31.3	31.1	–	–	–	0.075	158	154
FAEST-3s	[10]	–	0.90	73.2	75.7	–	–	–	0.164	3372	3282
FAEST-3f	[10]	–	0.90	73.1	75.7	–	–	–	0.225	363	354
FAEST-5s	[10]	–	1.02	143	148	–	–	–	0.672	5400	5252
FAEST-5f	[10]	–	1.02	143	148	–	–	–	0.692	712	694
MQOM-1s	Bal. [14]	–	2.65	13.5	16.8	–	–	–	9.23	291	194
MQOM-1f	Bal. [14]	–	2.50	9.50	10.7	–	–	–	9.95	73.2	47.9
MQOM-5s	Bal. [14]	–	6.95	35.1	40.7	–	–	–	118	2828	1656
MQOM-5f	Bal. [14]	–	6.68	30.4	32.2	–	–	–	124	710	459
MQOM-1s	Ref. [14]	–	142	1700	599	–	–	–	10.5	–	255
MQOM-1f	Ref. [14]	–	120	422	179	–	–	–	10.8	150	67.7
MQOM-5s	Ref. [14]	–	780	6550	1150	–	–	–	–	–	–
MQOM-5f	Ref. [14]	–	780	1180	827	–	–	–	–	–	–
MQOM-1s	Bal.^{1} [14]	–	2.64	27.5	28.2	–	–	–	9.23	381	249
MQOM-1f	Bal.^{1} [14]	–	2.50	34.4	35.0	–	–	–	9.95	88.2	56.4
MQOM-1s	Bal.^{2} [14]	–	2.64	17.0	20.4	–	–	–	9.23	459	457
MQOM-1f	Bal.^{2} [14]	–	2.50	21.5	21.6	–	–	–	9.95	101	63.1

Table 3: Code size, stack usage and timings: We report for the fast and short variant of each scheme. We consider the fastest variants of Mirath and MQOM with $\mathbb{F} = \mathbb{F}_{16}$. For MQOM and FAEST we only present numbers for their optimized implementation in Cortex-M4. The numbers for MQOM are taken from [14] and correspond to the five-round variant of MQOM. The last four rows concerning MQOM are for the OBT technique; The exponent in Bal. refers to the number of re-computations of the OBT. The numbers for FAEST are taken from [10] and concern its variant based on the Even-Mansour construction. The numbers for Mirath and RYDE on Cortex-M4 employ the reference implementation of the arithmetic related to the underlying problem, and are taken from their public implementations. The numbers in parentheses determines the savings with respect to the baseline implementation.

Cortex-M4 and, along with our novel constant-time Rijndael-256 implementation, ensure competitive performance, paving the way for practical TCitH deployments in embedded architectures. We believe that our framework should help cryptographers when proposing new TCitH-based schemes: our implementation is as generic as possible, so adding new constructions requires only implementing the underlying problem's mathematics and the polynomial proof procedures.

7.1 On Further Implications to MQOM

One takeaway from our results for MQOM is as follows. Notice that the OBT variant of MQOM illustrates a slower performance and bigger stack memory than its Forest$_\tau$ variant. Conversely, an implementation following our ZKP framework should provide faster performance and slightly less stack memory than the OBT variant of MQOM in [14]. We expect around the same performance as its Forest$_\tau$ variant since our ZKP framework ensures the same number of calculations as such a Forest$_\tau$ variant, but with a slightly bigger stack memory requirement.

However, if we allow the OBT to be re-computated at most twice, a modified implementation of the ZKP framework can achieve significantly lower stack requirements. More precisely, the idea is to modify the Build procedure to avoid the calculations of $\mathbf{p_x}$ and $\mathbf{p_u}$; instead, their calculation is referred to the step where the polynomial $\mathbf{p}_\alpha$ is determined. In particular, this approach avoids computing/storing all τ polynomials and computes them on the fly. Consequently, the stack memory required to store the polynomials $\mathbf{p_x}$ and $\mathbf{p_u}$ can be reduced by a factor of τ. A similar reasoning applies to the algorithm Rebuild. Furthermore, this 2-OBT approach is expected to require less stack memory and to be faster than the variant that recomputes OBT twice in [14].

Given the recent NIST announcement for the third round of the on-ramp competition for additional post-quantum signatures, and that the MQOM team possibly make changes according to [31,24,32], we limit our discussion to highlighting that the stack memory for this 2-OBT variant should be reduced as close to the signature sizes. For example, MQOM-1s should be able to reach below 3.5 KB of stack, while for MQOM-5s below 15 KB.

7.2 On Further Implications to SDitH

The stack memory situation in SDitH differs slightly from that of MQOM. The main difference is the large dimensions of the witnesses, which, for the current parameters of SDitH, are $n = 10360$, $n = 18396$, and $n = 19864$. Despite these sizes, our results should not be viewed negatively for SDitH, but rather as an opportunity for further optimizations in its implementation. Given the large witness dimensions, certain matrices and polynomials become too large to fit within constrained devices. To address this, one may adopt the 2-OBT approach as mentioned in Section 7.1. In addition, such matrices can be processed column-by-column or row-by-row in a streamlined fashion to considerably reduce stack memory usage. Nevertheless, further research is required to determine the minimum stack memory for SDitH, since non-convetional tricks may be needed to

reduce it further. We emphasize that the VOLEitH variant of SDitH is expected to require much more memory. Our findings could also be of interest for SDitH. Given the recent NIST announcement of the third round of the PQC competition and the possibility of SDitH updating parameters, we leave as future work an efficient implementation of SDitH for Cortex-M4 devices.

A Code listings for proposed ShiftRows variant

Listing 1.2: Our proposed ShiftRows variant for Rijndael-256 applied to s.

```
// Mask most significant halfword and bytewise rotate by 3
uint32_t t = ((s & 0x1F1F0000) << 3) | ((s & 0xE0E00000) >> 5);
// Combine with original least significant halfword
t |= s & 0x0000FFFF;
// Mask second and fourth bytes and bytewise rotate by 1
s = ((t & 0x7F007F00) << 1) | ((t & 0x80008000) >> 7);
// Combine with first and third bytes from previous rotation
s |= t & 0x00FF00FF;
```

Listing 1.3: Assembly code for our proposed ShiftRows variant for Rijndael-256.

```
    // Initialization (load bitmasks)
    mov r0, #0xF8F8F8F8
    mov r1, #0xFE00FE00

    // Apply ShiftRows once to r2, using r3 and r4 as temporaries
    and r3, r0, r2, lsl #3
    and r4, r2, #0xE0E0E0E0
    orr r3, r3, r4, lsr #5
    bfi r3, r2, #0, #16
    and r2, r1, r3, lsl #1
    and r4, r3, #0x80008000
    orr r2, r2, r4, lsr #7
    and r3, r3, #0x00FF00FF
    orr r2, r2, r3
```

References

1. Aaraj, N., Bettaieb, S., Bidoux, L., Budroni, A., Dyseryn, V., Esser, A., Feneuil, T., Gaborit, P., Kulkarni, M., Mateu, V., Palumbi, M., Perin, L., Rivain, M., Tillich, J.P., Xagawa, K.: PERK version 2.1. Tech. rep., National Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
2. Abdulrahman, A., Kannwischer, M.J., Lim, T.H.: Enabling microarchitectural agility: Taking ML-KEM & ML-DSA from cortex-M4 to M7 with SLOTHY. In: ASIACCS 25. pp. 1756–1771. ACM Press (Jun 2025). https://doi.org/10.1145/3708821.3736210

3. Aciiçmez, O., Schindler, W., Koç, Ç.: Cache based remote timing attack on the AES. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 271–286. Springer, Berlin, Heidelberg (Feb 2007). https://doi.org/10.1007/11967668_18
4. Adj, G., Aragon, N., Barbero, S., Bardet, M., Bellini, E., Bidoux, L., Chi-Domínguez, J.J., Dyseryn, V., Esser, A., Feneuil, T., Gaborit, P., Neveu, R., Rivain, M., Rivera-Zamarripa, L., Sanna, C., Tillich, J.P., Verbel, J., Zweydinger, F.: Mirath version 2.1. Tech. rep., National Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
5. Adomnicai, A.: An update on keccak performance on ARMv7-M. Cryptology ePrint Archive, Report 2023/773 (2023), https://eprint.iacr.org/2023/773
6. Adomnicai, A., Najm, Z., Peyrin, T.: Fixslicing: A new gift representation. IACR TCHES **2020**(3), 402–427 (2020). https://doi.org/10.13154/tches.v2020.i3.402-427, https://tches.iacr.org/index.php/TCHES/article/view/8595
7. Adomnicai, A., Peyrin, T.: Fixslicing AES-like ciphers. IACR TCHES **2021**(1), 402–425 (2021). https://doi.org/10.46586/tches.v2021.i1.402-425, https://tches.iacr.org/index.php/TCHES/article/view/8739
8. Advanced Encryption Standard (AES). National Institute of Standards and Technology, NIST FIPS PUB 197, U.S. Department of Commerce (Nov 2001)
9. Aragon, N., Bardet, M., Bidoux, L., Chi-Domínguez, J.J., Dyseryn, V., Feneuil, T., Gaborit, P., Joux, A., Neveu, R., Rivain, M., Tillich, J.P., Vinçotte, A.: RYDE version 2.1. Tech. rep., National Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
10. Aranha, D.F., Degn, J., Eilath, J., Nielsen, K., Scholl, P.: FAEST for memory-constrained devices with side-channel protections. Cryptology ePrint Archive, Report 2025/1261 (2025), https://eprint.iacr.org/2025/1261
11. Baum, C., Beullens, W., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: One tree to rule them all: Optimizing GGM trees and OWFs for post-quantum signatures. In: Chung, K.M., Sasaki, Y. (eds.) ASIACRYPT 2024, Part I. LNCS, vol. 15484, pp. 463–493. Springer, Singapore (Dec 2024). https://doi.org/10.1007/978-981-96-0875-1_15
12. Baum, C., Braun, L., Beullens, W., de Saint Guilhem, C.D., Klooß, M., Majenz, C., Mukherjee, S., Orsini, E., Ramacher, S., Rechberger, C., Roy, L., Scholl, P.: FAEST version 2.0. Tech. rep., National Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
13. Benadjila, R., Bouillaguet, C., Feneuil, T., Rivain, M.: MQOM version 2.1. Tech. rep., National Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
14. Benadjila, R., Feneuil, T.: Breaking the Myth of MPCitH Inefficiency: Optimizing MQOM for Embedded Platforms. Cryptology ePrint Archive, Report 2026/078 (2026), https://eprint.iacr.org/2026/078
15. Bernstein, D.J.: libcpucycles (2026), https://cpucycles.cr.yp.to
16. Bhadauria, R., Block, A.R., Ghosh, P., Thaler, J.: SNARK lower bounds via communication complexity. Cryptology ePrint Archive, Report 2025/1698 (2025), https://eprint.iacr.org/2025/1698
17. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE'97. LNCS, vol. 1267, pp. 260–272. Springer, Berlin, Heidelberg (Jan 1997). https://doi.org/10.1007/BFb0052352
18. Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M.: Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 235–251. Springer, Berlin, Heidelberg (Mar 2010). https://doi.org/10.1007/978-3-642-11925-5_17

19. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Berlin, Heidelberg (Oct 2006). https://doi.org/10.1007/11894063_16
20. Boyar, J., Matthews, P., Peralta, R.: Logic minimization techniques with applications to cryptology. Journal of Cryptology **26**(2), 280–312 (Apr 2013). https://doi.org/10.1007/s00145-012-9124-7
21. Companion Proof-of-Concept Code Implementation: The code is going to be public soon
22. Daemen, J., Rijmen, V.: The Design of Rijndael: The Advanced Encryption Standard (AES). Information Security and Cryptography, Springer-Verlag, Berlin (2020). https://doi.org/10.1007/978-3-662-60769-5
23. Drucker, N., Gueron, S.: Software optimization of rijndael for modern x86-64 platforms. In: Latifi, S. (ed.) ITNG 2022 19th International Conference on Information Technology-New Generations. pp. 147–153. Springer International Publishing, Cham (2022)
24. Feneuil, T., Rivain, M.: On the Security of MPC-in-the-head Signatures with Correlated GGM Trees. Cryptology ePrint Archive, Report 2026/615 (2026), https://eprint.iacr.org/2026/615
25. Huang, J., Adomnicai, A., Zhang, J., Dai, W., Liu, Y., Cheung, R.C.C., Koç, Ç.K., Chen, D.: Revisiting Keccak and Dilithium implementations on ARMv7-M. IACR TCHES **2024**(2), 1–24 (2024). https://doi.org/10.46586/tches.v2024.i2.1-24
26. Huth, J., Joux, A.: MPC in the head using the subfield bilinear collision problem. In: Reyzin, L., Stebila, D. (eds.) CRYPTO 2024, Part I. LNCS, vol. 14920, pp. 39–70. Springer, Cham (Aug 2024). https://doi.org/10.1007/978-3-031-68376-3_2
27. Kannwischer, M.J., Krausz, M., Petri, R., Yang, S.Y.: pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Report 2024/112 (2024), https://eprint.iacr.org/2024/112
28. Käsper, E., Schwabe, P.: Faster and timing-attack resistant AES-GCM. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 1–17. Springer, Berlin, Heidelberg (Sep 2009). https://doi.org/10.1007/978-3-642-04138-9_1
29. Kim, S., Lee, B., Son, M.: Relaxed vector commitment for shorter signatures. In: Fehr, S., Fouque, P.A. (eds.) EUROCRYPT 2025, Part IV. LNCS, vol. 15604, pp. 427–455. Springer, Cham (May 2025). https://doi.org/10.1007/978-3-031-91134-7_15
30. Kohlweiss, M., Pancholi, M., Takahashi, A.: How to compile polynomial IOP into simulation-extractable SNARKs: A modular approach. In: Rothblum, G.N., Wee, H. (eds.) TCC 2023, Part III. LNCS, vol. 14371, pp. 486–512. Springer, Cham (Nov / Dec 2023). https://doi.org/10.1007/978-3-031-48621-0_17
31. Kosuge, H., Xagawa, K.: New security proofs of MPC-in-the-head signatures in the quantum random oracle model. Cryptology ePrint Archive, Report 2025/1999 (2025), https://eprint.iacr.org/2025/1999
32. Kosuge, H., Xagawa, K.: Towards Formal Security Proofs of MQOM. Cryptology ePrint Archive, Report 2026/629 (2026), https://eprint.iacr.org/2026/629
33. Matsui, M., Nakajima, J.: On the power of bitslice implementation on intel core2 processor. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 121–134. Springer, Berlin, Heidelberg (Sep 2007). https://doi.org/10.1007/978-3-540-74735-2_9
34. Melchor, C.A., Bettaieb, S., Bidoux, L., Feneuil, T., Gaborit, P., Gama, N., Gueron, S., Howe, J., H ulsing, A., Joseph, D., Joux, A., Kulkarni, M., Persichetti, E., Randrianarisoa, T.H., Rivain, M., Yue, D.: SDitH version 2.0. Tech. rep., National

Institute of Standards and Technology (2025), https://csrc.nist.gov/projects/pqc-dig-sig/round-2-additional-signatures
35. National Institute of Standards and Technology: NIST SP 800-197 (initial preliminary draft) pre-draft call for comments: NIST proposes to standardize a wider variant of AES (Dec 2024), https://csrc.nist.gov/pubs/sp/800/197/iprd
36. Petri, R.: Comment to issue #180 in the pqm4 GitHub repository: "Add an option to use the CCM as main RAM (or at least as stack)" (2021), https://github.com/mupq/pqm4/issues/180#issuecomment-775981267
37. Pornin, T.: BearSSL: a smaller SSL/TLS library (2026), https://www.bearssl.org/
38. Rodrigues, F.C., Filho, D.L.G., Adj, G., Martinez, I.A.C., Chávez-Saab, J., López-Hernández, J.C., Scott, M., Rodríguez-Henríquez, F.: Generation of fast finite field arithmetic forCortex-M4 with ECDH and SQIsign applications. IACR TCHES **2025**(4), 588–620 (2025). https://doi.org/10.46586/tches.v2025.i4.588-620
39. Saarinen, M.J.O.: Brief comments on rijndael-256 and the standard RISC-V cryptography extensions. Cryptology ePrint Archive, Report 2025/1198 (2025), https://eprint.iacr.org/2025/1198
40. Schwabe, P., Stoffelen, K.: All the AES you need on Cortex-M3 and M4. In: Avanzi, R., Heys, H.M. (eds.) SAC 2016. LNCS, vol. 10532, pp. 180–194. Springer, Cham (Aug 2016). https://doi.org/10.1007/978-3-319-69453-5_10