Mirath Signature Scheme

25/07/2025

Submitters (alphabetical order):

- Gora Adj (Technology Innovation Institute, UAE)
- Nicolas Aragon (Naquidis Center, FR)
- Stefano Barbero (Politecnico di Torino, IT)
- Magali Bardet (LITIS, University of Rouen Normandie & INRIA, FR)
- Emanuele Bellini (Technology Innovation Institute, UAE)
- Loïc Bidoux (Technology Innovation Institute, UAE)
- Jesús-Javier Chi-Domínguez (Technology Innovation Institute, UAE)
- Victor Dyseryn (Télécom Paris, FR)
- Andre Esser (Technology Innovation Institute, UAE)
- Thibauld Feneuil (Cryptoexperts, FR)
- Philippe Gaborit (University of Limoges, FR)
- Romaric Neveu (University of Limoges, FR)
- Matthieu Rivain (Cryptoexperts, FR)
- Luis Rivera-Zamarripa (Technology Innovation Institute, UAE)
- Carlo Sanna (Politecnico di Torino, IT)
- Jean-Pierre Tillich (INRIA, FR)
- Javier Verbel (Technology Innovation Institute, UAE)
- Floyd Zweydinger (Technology Innovation Institute, UAE)

Contact: team@pqc-mirath.org

Version: 2.0.1

Table of Contents

List of Tables

List of Algorithms

Changelog

Version 2.0.1 (25/07/2025)

- The implementations of Mirath have been improved (performance speed-up and fixing constant-time issues). The constant-time claim regarding Mirath implementations has been restricted to its optimized implementation.

Version 2.0.0 (05/02/2025)

- The design of Mirath relies on the TCitH framework [25,23] (or alternatively the VOLEitH framework) along with the Dual Support Modeling [12]. As a result, Mirath signature sizes have been significantly reduced with respect to the signature sizes of MIRA and MiRitH.

- Mirath results from the merge of the MIRA [3,4] and MiRitH [1,2] schemes.

1 Introduction

Mirath is a post-quantum digital signature scheme based on the hardness of the MinRank problem. Informally, the MinRank problem [13] asks to find a non-trivial low-rank linear combination of some given matrices over a finite field (actually, Mirath employs an equivalent "dual version" of this problem [12]).

Mirath is based on a Zero-Knowledge Proof of Knowledge (ZKPoK) of a solution to an instance of the MinRank problem. This ZKPoK is designed following the Multi-Party-Computation-in-the-Head (MPCitH) paradigm [30]. More precisely, we rely on the Threshold-Computation-in-the-Head (TCitH) [22,24] framework to build Mirath, while a variant is possible using VOLE-in-the-Head (VOLEitH) [9] framework. The ZKPoK is then converted into a signature scheme using the Fiat–Shamir transform [26].

Mirath has been developed by the merger of two teams, each of which had previously devised its own post-quantum digital signature scheme based on the hardness of the MinRank problem, namely MIRA [3,4] and MiRitH [1,2]. The name "Mirath" itself is composed from the names "MIRA" and "MiRitH."

This document is structured as follows. We describe some mathematical background and the MinRank problem in Section 2. We provide a high-level description of Mirath in Section 3. Section 4 is devoted to the detailed description of all the algorithms used in the signature scheme. The parameter sets of Mirath and their corresponding public-key and signature sizes are given in Section 5. A performance analysis of the scheme is presented in Section 6. A security analysis of Mirath and some known attacks are provided in Section 7 and Section 8, respectively. Finally, in Section 9, we summarize the main advantages and limitations of Mirath.

2 Mathematical Background and Notations

2.1 Notations

The main mathematical notations employed throughout this paper are summarized in Table 1. We employ lower-case and upper-case bold letters for (column) vectors and matrices, respectively. For every $n_r \times n_c$ matrix $\boldsymbol{M}$, we let $\mathrm{vec}(\boldsymbol{M})$ denotes the column-major vectorization of $\boldsymbol{M}$, that is, the $n_r n_c \times 1$ column vector whose entries are the entries of $\boldsymbol{M}$ in column-major order.

Symbol	Meaning
$\{0,1\}^{\ell}$	Set of binary strings of length ℓ.
$\{0,1\}^{*}$	Set of binary strings of finite length.
$\mathsf{str}_1 \parallel \mathsf{str}_2$	Concatenation of the binary strings str_1 and str_2.
$\mathbb{F}_q$	A finite field of q elements.
$\mathbb{F}_q^{n_c \times n_r}$	The vector space of $n_c \times n_r$ matrices over the field $\mathbb{F}_q$.
$\mathbf{0}_{n_c \times n_r}$	The $n_c \times n_r$ zero matrix.
$\boldsymbol{I}_s$	The $s \times s$ identity matrix.
$\mathrm{rank}(\boldsymbol{M})$	The rank of the matrix $\boldsymbol{M}$.
$\mathrm{vec}(\boldsymbol{M})$	columns-major vectorization of the matrix $\boldsymbol{M}$.
$(\boldsymbol{A} \mid \boldsymbol{B})$	The matrix obtained by juxtaposing the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.
log	The base-2 logarithm.
$\otimes$	Bitwise multiplication (AND).
$\oplus$	Bitwise addition (XOR).

Table 1. Mathematical notation.

2.2 MinRank Problem

The MinRank problem is the underlying hard problem of Mirath.

Definition 1 (MinRank Problem). *Let q, m, n, k, and r be positive integers. Let $\boldsymbol{M}_1, \ldots, \boldsymbol{M}_k, \boldsymbol{E} \in \mathbb{F}_q^{m \times n}$ and $\boldsymbol{x} := (x_1, \ldots, x_k) \in \mathbb{F}_q^k$ be uniformly sampled such that*

$$\mathsf{rank}(\boldsymbol{E}) \leq r \quad \textit{and} \quad \boldsymbol{M} := \boldsymbol{E} - \sum_{i=1}^{k} x_i \boldsymbol{M}_i.$$

Given $\boldsymbol{M}, \boldsymbol{M}_1 \ldots, \boldsymbol{M}_k$, the MinRank problem of parameters (q, m, n, k, r) asks to retrieve the vector $\boldsymbol{x}$.

Usually, the instance of the MinRank problem is generated by uniformly sampling the matrices $\boldsymbol{M}, \boldsymbol{M}_1, \ldots, \boldsymbol{M}_k$ so that $\mathrm{rank}\left(\boldsymbol{M} + \sum_{i=1}^{k} x_i \boldsymbol{M}_i\right) \leq r$. Actually, in order to have a solution with a smaller size, Mirath employs the following equivalent version of the MinRank problem, namely, the MinRank Syndrome Problem, which was introduced by Bidoux et al. [12, Definition 8 and Proposition 2]. To introduce this version of the problem, we will use the application vec. In fact, this application allows to obtain the generator matrix of the code $\langle \boldsymbol{M}_1, \ldots, \boldsymbol{M}_k \rangle$ as the matrix $\boldsymbol{G} \in \mathbb{F}_q^{mn \times k}$ whose columns are $\mathrm{vec}(\boldsymbol{M}_1), \ldots, \mathrm{vec}(\boldsymbol{M}_k)$, i.e,

$$\boldsymbol{G} = \left(\mathrm{vec}(\boldsymbol{M}_1) \ldots \mathrm{vec}(\boldsymbol{M}_k)\right) \ .$$

By using the dual matrix of the code generated by $\boldsymbol{G}$, i.e, using $\boldsymbol{H} \in \mathbb{F}_q^{(mn-k) \times mn}$ such that $\boldsymbol{HG} = \boldsymbol{0}$, the MinRank Syndrome Problem, on which Mirath relies, can be obtained as follows:

Definition 2 (MinRank Syndrome Problem). *Let q, m, n, k and r be positive integers. Let $\boldsymbol{H} := \left[\boldsymbol{I}_{mn-k} \ \boldsymbol{H'}\right] \in \mathbb{F}_q^{(mn-k) \times mn}$ where $\boldsymbol{H'} \in \mathbb{F}_q^{(mn-k) \times k}$ is a uniformly sampled matrix, and $\boldsymbol{y} \in \mathbb{F}_q^{mn-k}$. The computational MinRank Syndrome Problem(q, m, n, k, r) asks to find $\boldsymbol{E}$ such that*

$$\boldsymbol{H}\,\mathrm{vec}(\boldsymbol{E}) = \boldsymbol{y} \quad \textit{and} \quad \mathsf{rank}(\boldsymbol{E}) \leq r.$$

2.3 All-but-one Vector Commitment (AVC)

The Mirath scheme rely on an *all-but-one vector commitment scheme*. Such a primitive enables to commit to N *random* values $v_1, \ldots, v_N$ to later reveal all of them except one. While there exists a naive solution which would consist in committing each value independently and reveal all of them except one, an AVC scheme aims to propose a more efficient solution, for which the communication cost is sublinear in N (instead of being linear in N as in the naive solution). In what follows, we give the definition of an AVC scheme. We only consider the case where the random values are λ-bit seeds, since it is the setting which is useful for the Mirath scheme.

Definition 3 (All-but-one Vector Commitment (AVC)). *An all-but-one vector commitment (AVC) scheme with message space $\{0,1\}^\lambda$ and parameter N is defined by the following algorithms:*

- $\mathsf{Commit}() \to (h_{com}, \mathsf{key}, (\mathsf{seed}_1, \ldots, \mathsf{seed}_N))$*: Output a commitment h_{com} with opening key key for messages $(\mathsf{seed}_1, \ldots, \mathsf{seed}_N) \in \left(\{0,1\}^\lambda\right)^N$.*
- $\mathsf{Open}(\mathsf{key}, i^*)$*: On a opening key key and an index $i^* \in \{1, \ldots, N\}$, output an opening π_{AVC} for $\{\mathsf{seed}_i\}_{i \neq i^*}$.*
- $\mathsf{Reconstruct}(\pi_{AVC}, i^*) \to (\{\mathsf{seed}_i\}_{i \neq i^*}, h_{com})$*: On the opening π_{AVC} and an index $i^* \in \{1, \ldots, N\}$, output the messages $\{\mathsf{seed}_i\}_{i \neq i^*}$, together the commitment h_{com}.*

The scheme is said complete *if an honestly-generated opening leads to the same messages and the same commitment, namely for all i^*, if*

$$(h_{com}, \mathsf{key}, (\mathsf{seed}_1, \ldots, \mathsf{seed}_N)) \leftarrow \mathsf{Commit}()$$
$$\pi_{AVC} \leftarrow \mathsf{Open}(\mathsf{key}, i^*)$$

then we have $h'_{com} = h_{com}$ and $\mathsf{seed}'_i = \mathsf{seed}_i$ for all $i \neq i^*$, where

$$\left((\mathsf{seed}'_1, \ldots, \mathsf{seed}'_N), h'_{com}\right) \leftarrow \mathsf{Reconstruct}(\pi_{AVC}, i^*).$$

The scheme is said binding *if it is hard to produce two openings π_{AVC} and π'_{AVC} leading to the same commitment while having differents messages. Finally, the scheme is said* hiding *if it hard to recover seed_{i^*} from π_{AVC}.*

Let us remark that there are alternative definitions for the AVC scheme. For example, instead of having the Reconstruct routine, we might have a Verify routine which takes h_{com} and π_{AVC} as input and outputs the messages when the opening is consistent with the commitment and $\perp$ otherwise. However, in a context of signature scheme, the Reconstruct routine is more convenient.

In the Mirath scheme, we will need to use τ AVC schemes in parallel. Therefore, instead of considering τ independent commitments, we can use a batched variant of the all-but-one vector commitment scheme, which aims to produce smaller opening proof (an opening proof of a batched AVC will be smaller than τ opening proofs of an AVC).

Definition 4 (Batched All-but-one Vector Commitment (BAVC)). *A batched all-but-one vector commitment (BAVC) scheme with message space $\{0,1\}^\lambda$ and parameters (N, τ) is defined by the following algorithms:*

- $\mathsf{Commit}() \to (h_{com}, \mathsf{key}, \{(\mathsf{seed}_1^{(e)}, \ldots, \mathsf{seed}_N^{(e)})\}_{e\in[1,\tau]})$*: Output a commitment h_{com} with opening key key for messages $(\mathsf{seed}_1^{(e)}, \ldots, \mathsf{seed}_N^{(e)}) \in \left(\{0,1\}^\lambda\right)^N$ for all $e \in [1, \tau]$.*
- $\mathsf{Open}(\mathsf{key}, \{i^{*(e)}\}_e)$*: On a opening key key and τ indexes $i^{*(e)} \in \{1, \ldots, N\}$, output an opening π_{BAVC} for $\{\mathsf{seed}_i^{(e)}\}_{e,i\neq i^{*(e)}}$.*
- $\mathsf{Reconstruct}(\pi_{\mathsf{BAVC}}, \{i^{*(e)}\}_e) \to \left(\{\mathsf{seed}_i^{(e)}\}_{e,i\neq i^{*(e)}}, h_{com}\right)$*: On the opening π_{BAVC} and indexes $i^{*(e)} \in \{1, \ldots, N\}$, output the messages $\{\mathsf{seed}_i^{(e)}\}_{e,i\neq i^{*(e)}}$, together the commitment h_{com}.*

The properties of a BAVC *scheme are the same than those of an AVC scheme, up to the fact that they are defined for the τ parallel commitments.*

3 High-Level Description of Mirath

In this section, we provide a high-level description of the Mirath signature scheme. This scheme is built by applying the Fiat–Shamir transform on top of a ZKPoK of a solution to an instance of the MinRank problem. The underlying proof system uses the TCitH framework [25,24]. Parameters corresponding to a variant using the VOLEitH framework [9] are given in Appendix A.

3.1 TCitH Framework in the PIOP Formalism

The MPCitH paradigm [30] is a versatile method introduced in 2007 to build zero-knowledge proof systems using techniques from secure multi-party computation (MPC). This paradigm has been drastically improved in recent years and is particularly efficient to build zero-knowledge proofs for small circuits such as those involved in (post-quantum) signature schemes. The more recent MPCitH-based frameworks are the VOLE-in-the-Head (VOLEitH) framework from [9] and the Threshold-Computation-in-the-Head (TCitH) framework from [25,24].

In this subsection, we will describe the general proof system on which Mirath is relying on. In what follows, we present this proof system using the formalism of the Polynomial Interactive Oracle Proofs (PIOP), as presented by Feneuil [21]. Indeed, while the TCitH and VOLEitH frameworks have been respectively introduced using a sharing-based and a VOLE-based formalism, one can unify those two frameworks using the PIOP formalism, which enables us to have a description that does not depend on MPC technologies[1], leading to an easier-to-understand scheme for those who do not already know those two frameworks.

Let us assume that we want to build an interactive zero-knowledge proof that would enable a prover to convince a verifier that the prover knows a witness $w \in \mathbb{F}_q^n$ that satisfies some public polynomial relations:

$$f_j(w) = 0, \quad \text{for all } j \in \{1, \dots, m\},$$

where $f_1, \dots, f_m$ are polynomials over $\mathbb{F}_q$ of total degree at most d. Let us consider two proof parameters $N, \mu \in \mathbb{N}$ such that $N \leq 2^\mu$. The proof system we consider is the following:

1. For each $j \in \{1, \dots, n\}$, the prover samples a random degree-1 polynomials P_j such that $P_j(X) = w_j \cdot X + (w_{\mathsf{base}})_j$ for some $(w_{\mathsf{base}})_j \in \mathbb{F}_{q^\mu}$. He also samples a random degree-$(d-1)$ polynomial $P_0 \in \mathbb{F}_{q^\mu}[X]$. He commits to those polynomials.

[1] In the TCitH framework, instead of performing operations over Shamir's secret sharings, we can directly work over their underlying polynomials. In the VOLEitH framework, instead of performing operations over VOLE gadgets, we can directly work over their underlying degree-1 polynomials.

2. The verifier chooses random coefficients $\gamma_1, \ldots, \gamma_m \in \mathbb{F}_{q^\mu}$ and sends them to the prover. The latter then reveals the degree-$(d-1)$ polynomial $Q(X)$ defined as

$$Q(X) = P_0(X) + \sum_{j=1}^{m} \gamma_j \cdot f_j^{[h]}(X, P_1(X), \ldots, P_n(X)), \tag{1}$$

   where $f_j^{[h]}$ is the homogeneous version of the polynomial f_j, *i.e.*

$$f_j^{[h]}(Y, X_1, \ldots, X_n) := Y^d \cdot f_j(X_1/Y, \ldots, X_n/Y).$$

3. The verifier samples a random evaluation point r from a public subset $S \subset \mathbb{F}_{q^\mu}$ of cardinality N and sends it to the prover. The latter then reveals the evaluations $v_i := P_i(r)$, together with a proof π that the evaluations are consistent with the commitment.
4. The verifier checks that the revealed evaluations are consistent with the commitment using π and checks that we have

$$Q(r) = v_0 + \sum_{j=1}^{m} \gamma_j \cdot f_j^{[h]}(r, v_1, \ldots, v_n) \ . \tag{2}$$

The above protocol assumes that the prover has a way to commit polynomials and to provably open some evaluations later (while keeping hidden the other evaluations).

Security Analysis. We can observe that the coefficient in front of the degree-d monomial in the right term of Equation (1) is

$$\sum_{j=1}^{m} \gamma_j \cdot f_j(w_1, \ldots, w_n) \ , \tag{3}$$

so the degree-$(d-1)$ polynomial Q is well-defined because this quantity is zero when the witness w is valid. Let us assume that the prover is malicious, meaning that he does not have a valid witness. This implies that there exists j^* such that $f_{j^*}(w) \neq 0$. In such a case, the probability that there exists some Q such that Eq. (1) holds is at most $1/q^\mu$ over the randomness of $\gamma_1, \ldots, \gamma_m$, because the coefficient (3) is zero only with probability $1/q^\mu$. If Eq. (1) does not hold, the probability that the check in Eq. (2) passes is at most d/N, since the degree-d polynomial relation

$$Q(X) - \left(P_0 + \sum_{j=1}^{m} \gamma_j \cdot f_j^{[h]}(X, P_1(X), \ldots, P_m(X)) \right) \neq 0$$

would have at most d roots (and so the random challenge r should be among those roots). Thus, the proof system is *sound*, with a soundness error of

$$\frac{1}{q^\mu} + \left(1 - \frac{1}{q^\mu}\right) \cdot \frac{d}{N}.$$

Moreover, assuming that the commitment scheme is hiding, we can observe that the interactive proof is zero-knowledge since

- revealing $Q(X)$ leaks no information about the secret thanks to the random polynomial P_0, and
- revealing one evaluation of the polynomials $P_1, \ldots, P_n$ leaks no information about the leading term thanks to the randomness used to build those polynomials.

In our setting of the Mirath signature scheme, $\mathbb{F}_{q^\mu}$ might be a small field, so it would lead to a relatively bad soundness error. To solve this issue, we just repeat Step 2 of the proof system ρ times in parallel: the verifier chooses a random matrix $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times m}$ and then the prover reveals ρ polynomials $Q_1, \ldots, Q_\rho$ such that

$$Q_k(X) = P_{0,k}(X) + \sum_{j=1}^{m} \boldsymbol{\Gamma}_{k,j} \cdot f_j^{[h]}(X, P_1(X), \ldots, P_n(X))$$

where $P_{0,1}, \ldots, P_{0,\rho}$ are ρ random degree-$(d-1)$ polynomials that have been committed in the previous step. In that case, the soundness error is now

$$\frac{1}{q^{\mu \cdot \rho}} + \left(1 - \frac{1}{q^{\mu \cdot \rho}}\right) \cdot \frac{d}{N}.$$

Remark 1. Using the above tweak, when taking ρ such that $\rho \cdot \mu \cdot \log_2 q \geq \lambda$, the soundness error is roughly d/N. When N is small (compared to 2^λ), we would need to repeat the zero-knowledge protocol several times to achieve the desired security level. One solution could be that the verifier checks the polynomial relation (1) into several points instead of a single one. If the verifier checks the relation into ℓ points, the prover needs to sample $P_1, \ldots, P_n$ as degree-ℓ polynomials to preserve zero-knowledge, and the soundness error would be

$$\frac{1}{q^{\mu \cdot \rho}} + \left(1 - \frac{1}{q^{\mu \cdot \rho}}\right) \cdot \frac{\binom{d \cdot \ell}{\ell}}{\binom{N}{\ell}}.$$

By taking ℓ such the soundness error is directly negligible, we would not need to rely on parallel repetitions. However, the techniques we would like to use to commit to polynomials are based on GGM trees and do not scale well if we want to open several evaluations. We would need to use techniques based on Merkle trees (*e.g.* TCitH-MT [24]), but the signature size would be larger than 6 kB (for the first security level).

In what follows, we describe how to commit polynomials such that we can later open some evaluations.

The TCitH-GGM Approach. The TCitH framework [24] shows that we can commit $\bar{n}$ random polynomials using seed trees thanks to ideas from [31,15]. Here is the commitment process for degree-1 polynomials:

1. One uses an all-but-one vector commitment (AVC) to sample and commit N seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_N$.
2. One expands each seed_i as $w_{\mathsf{rnd},i} := \mathsf{PRG}(\mathsf{seed}_i) \in \mathbb{F}_q^{\bar{n}}$ for $i \in \{1, \ldots, N\}$, where PRG is a pseudorandom generator.
3. One computes

$$w_{\mathsf{acc}} \leftarrow \sum_{i=1}^{N} w_{\mathsf{rnd},i} \in \mathbb{F}_q^{\bar{n}}$$
$$w_{\mathsf{base}} \leftarrow -\sum_{i=1}^{N} \phi(i) \cdot w_{\mathsf{rnd},i} \in \mathbb{F}_{q^\mu}^{\bar{n}}$$

   where $\phi : \{1, \ldots, N\} \to \mathbb{F}_{q^\mu}$ is a public one-to-one function.
4. One reveals the auxiliary value $\mathsf{aux} := w - w_{\mathsf{acc}}$.
5. One defines P_j as

$$P_j(X) = w_j \cdot X + (w_{\mathsf{base}})_j$$

   for all j.

This commitment procedure has the main advantage to enable the prover to reveal one evaluation $\{P_j(\phi(i^*))\}_j$ for $i^* \in \{1, \ldots, N\}$ while keeping *secret* the coefficient w and w_{base}: they just need to reveal all the $\{\mathsf{seed}_i\}_i$ except seed_{i^*} (by opening the AVC scheme) and the verifier will be able to compute $P_j(\phi(i^*))$ as

$$\phi(i^*) \cdot \mathsf{aux}_j + \sum_{i=1, i \neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot (w_{\mathsf{rnd},i})_j \quad \text{with} \quad w_{\mathsf{rnd},i} := \mathsf{PRG}(\mathsf{seed}_i).$$

Indeed, we have that

$$\begin{aligned}
\phi(i^*) \cdot \mathsf{aux}_j + & \sum_{i=1, i \neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot (w_{\mathsf{rnd},i})_j \\
&= \phi(i^*) \cdot \left(\mathsf{aux}_j + \sum_{i=1}^{N} (w_{\mathsf{rnd},i})_j \right) - \sum_{i=1}^{N} \phi(i) \cdot (w_{\mathsf{rnd},i})_j \\
&= \phi(i^*) \cdot (\mathsf{aux}_j + (w_{\mathsf{acc}})_j) + (w_{\mathsf{base}})_j \\
&= \phi(i^*) \cdot w_j + (w_{\mathsf{base}})_j = P_j(\phi(i^*))
\end{aligned}$$

Using this commitment procedure, the zero-knowledge protocol has 5 rounds, and one needs to rely on protocol repetitions to achieve the desired security. Indeed, the computational complexity is linear in N and so we can not take N exponentially large. To have a λ-bit security we need to repeat the protocol τ times in parallel, such that $(d/N)^\tau \leq 2^{-\lambda}$, assuming that $q^{-\mu \cdot \rho}$ is negligible.

The VOLEitH Approach. As the TCitH framework, the VOLEitH approach starts by committing τ sets of polynomials $\{P_i^{(1)}\}_i, \ldots, \{P_i^{(\tau)}\}_i$ in parallel. However, instead of considering those sets of polynomials individually as the TCitH

framework, the VOLEitH framework [9] consists in "merging them" into polynomials for which we will be able to open N^τ evaluations. This merge introduces an additional round in the proof system, leading to a 7-round proof system with soundness error

$$\frac{1}{2^{\mu\cdot\rho}} + \left(1 - \frac{1}{2^{\mu\cdot\rho}}\right) \cdot \frac{d}{N^\tau} \ .$$

This approach is used to construct a variant of Mirath described in Appendix A.

3.2 Dual Support Modeling

The Mirath signature scheme is built from a zero-knowledge proof of knowledge for a solution to a MinRank instance. We rely on the proof system described in the previous section. This scheme enables us to prove the knowledge of a witness satisfying some degree-d polynomial constraints, so we should write the MinRank problem as a system of degree-d polynomial equations.

Instead of relying on the standard MinRank problem, Mirath uses the Dual Support Modeling from [12], which relies on the equivalent MinRank Syndrome problem. In this setting, the protocol aims at verifying that a matrix $\boldsymbol{E}$ is solution of the constraints

$$\begin{cases} \boldsymbol{H}\,\mathrm{vec}(\boldsymbol{E}) = \boldsymbol{y}, \\ \mathsf{rank}(\boldsymbol{E}) \leq r \end{cases}$$

for a given matrix $\boldsymbol{H} \in \mathbb{F}_q^{(mn-k)\times mn}$ and a given vector $\boldsymbol{y} \in \mathbb{F}_q^{mn-k}$, where $\mathrm{vec}(\cdot)$ returns a flatten version of the input matrix. The modeling consists in viewing $\boldsymbol{E}$ as a product of two matrices, $\boldsymbol{E} = \boldsymbol{S}\boldsymbol{C}$, with $\boldsymbol{S} \in \mathbb{F}_q^{m\times r}$ and $\boldsymbol{C} \in \mathbb{F}_q^{r\times n}$. Furthermore, the modeling specializes the matrix $\boldsymbol{C}$ as $\left[\boldsymbol{I}_r\ \boldsymbol{C'}\right]$ for some matrix $\boldsymbol{C'} \in \mathbb{F}_q^{r\times(n-r)}$. Therefore, we use the proof system to prove that we know the witness $(\boldsymbol{S}, \boldsymbol{C'})$ which satisfy the following *quadratic* constraints ($d = 2$):

$$\boldsymbol{H}\,\mathrm{vec}(\boldsymbol{S}\boldsymbol{C}) = \boldsymbol{y} \quad \text{with} \quad \boldsymbol{C} = \left[\boldsymbol{I}_r\ \boldsymbol{C'}\right].$$

3.3 Mirath Protocol

Let us now express the proof system in the specific case of the Mirath scheme, *i.e.* specialize the proof system described in Section 3.1 for the MinRank modeling of Section 3.2:

1. The prover begins by sampling random degree-1 polynomials $P_S = \boldsymbol{S}\cdot X + \boldsymbol{S}_{\mathsf{base}}$ and $P_{C'} = \boldsymbol{C'}\cdot X + \boldsymbol{C'}_{\mathsf{base}}$, where $\boldsymbol{S}_{\mathsf{base}}$ and $\boldsymbol{C'}_{\mathsf{base}}$ are randomly sampled from $\mathbb{F}_{q^\mu}^{m\times r}$ and $\mathbb{F}_{q^\mu}^{r\times(n-r)}$ respectively. He also samples a random degree-1 polynomial $P_v = \boldsymbol{v}\cdot X + \boldsymbol{v}_{\mathsf{base}}$ where $\boldsymbol{v} \in \mathbb{F}_{q^\mu}^{\rho}$ and $\boldsymbol{v}_{\mathsf{base}} \in \mathbb{F}_{q^\mu}^{\rho}$. He commits to those polynomials.
2. The verifier chooses a random matrix $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho\times(mn-k)}$ and sends it to the prover. The latter reveals the degree-1 polynomial $P_\alpha(X)$ defined as

$$P_\alpha(X) = P_v(X) + \boldsymbol{\Gamma}\cdot\left(\boldsymbol{H}\cdot\mathrm{vec}(P_E(X)) - \boldsymbol{y}\cdot X^2\right) \in (\mathbb{F}_{q^\mu}[X])^\rho$$

with

$$P_E(X) = P_S(X) \cdot [P_{I_r}(X) \parallel P_{C'}] \in (\mathbb{F}_{q^\mu}[X])^{m \times n} \ ,$$

where $P_{I_r} := \boldsymbol{I}_r \cdot X \in (\mathbb{F}_{q^\mu}[X])^{r \times r}$.

3. The verifier samples a random evaluation point r from a public subset $S \subset \mathbb{F}_{q^\mu}$ of size N and sends it to the prover. The latter then reveals the evaluations $\boldsymbol{S}_{\mathsf{eval}} := P_S(r)$, $\boldsymbol{C'}_{\mathsf{eval}} := P_{C'}(r)$ and $\boldsymbol{v}_{\mathsf{eval}} := P_v(r)$, together with a proof π that the evaluations are consistent with the commitment.
4. The verifier checks that the revealed evaluations are consistent with the commitment using π and check that we have

$$P_\alpha(r) =_? \boldsymbol{v}_{\mathsf{eval}} + \boldsymbol{\Gamma} \cdot \left(\boldsymbol{H} \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{eval}}) - \boldsymbol{y} \cdot r^2 \right) \in \mathbb{F}_{q^\mu}^{\rho}$$

with

$$\boldsymbol{E}_{\mathsf{eval}} = \boldsymbol{S}_{\mathsf{eval}} \cdot [r \cdot \boldsymbol{I}_r \parallel \boldsymbol{C'}_{\mathsf{eval}}].$$

Committing to Witness Polynomials. In the above proof system, we need to commit three polynomials:

- The witness polynomial $P_S = \boldsymbol{S} \cdot X + \boldsymbol{S}_{\mathsf{base}}$ encoding the secret $\boldsymbol{S}$;
- The witness polynomial $P_{C'} = \boldsymbol{C'} \cdot X + \boldsymbol{C'}_{\mathsf{base}}$ encoding the secret $\boldsymbol{C'}$;
- The masking polynomial $P_v = \boldsymbol{v} \cdot X + \boldsymbol{v}_{\mathsf{base}}$, which aims to avoid leakage through P_α.

To commit them, we use the TCitH approach as explained in Section 3.1:

1. We use an all-but-one vector commitment (AVC) to sample and commit N seeds $\mathsf{seed}_1, \ldots, \mathsf{seed}_N$.
2. One expands each seed_i as

$$(\boldsymbol{S}_{\mathsf{rnd},i}, \boldsymbol{C'}_{\mathsf{rnd},i}, \boldsymbol{v}_{\mathsf{rnd},i}) := \mathsf{PRG}(\mathsf{seed}_i) \in \mathbb{F}_q^{m \times r} \times \mathbb{F}_q^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$$

for all $i \in \{1, \ldots, N\}$.

3. One computes

$$(\boldsymbol{S}_{\mathsf{acc}}, \boldsymbol{C'}_{\mathsf{acc}}, \boldsymbol{v}_{\mathsf{acc}}) \leftarrow \left(\sum_{i=1}^{N} \boldsymbol{S}_{\mathsf{rnd},i}, \sum_{i=1}^{N} \boldsymbol{C'}_{\mathsf{rnd},i}, \sum_{i=1}^{N} \boldsymbol{v}_{\mathsf{rnd},i} \right)$$

$$(\boldsymbol{S}_{\mathsf{base}}, \boldsymbol{C'}_{\mathsf{base}}, \boldsymbol{v}_{\mathsf{base}}) \leftarrow \left(-\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{S}_{\mathsf{rnd},i}, -\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{C'}_{\mathsf{rnd},i}, -\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{v}_{\mathsf{rnd},i} \right)$$

with $(\boldsymbol{S}_{\mathsf{acc}}, \boldsymbol{C'}_{\mathsf{acc}}, \boldsymbol{v}_{\mathsf{acc}}) \in \mathbb{F}_q^{m \times r} \times \mathbb{F}_q^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$ and $(\boldsymbol{S}_{\mathsf{base}}, \boldsymbol{C'}_{\mathsf{base}}, \boldsymbol{v}_{\mathsf{base}}) \in \mathbb{F}_{q^\mu}^{m \times r} \times \mathbb{F}_{q^\mu}^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$.

4. One reveals the auxiliary value $\mathsf{aux} := (\boldsymbol{S}_{\mathsf{aux}}, \boldsymbol{C'}_{\mathsf{aux}})$ with $\boldsymbol{S}_{\mathsf{aux}} := \boldsymbol{S} - \boldsymbol{S}_{\mathsf{acc}}$ and $\boldsymbol{C'}_{\mathsf{aux}} = \boldsymbol{C'} - \boldsymbol{C'}_{\mathsf{acc}}$). Since $\boldsymbol{v}$ is a random vector (which is not part of the witness, it aims to mask the polynomial P_α), we do not need to rely on auxiliary value, we just define $\boldsymbol{v}$ as $\boldsymbol{v}_{\mathsf{acc}}$ (it is equivalent to say that $\boldsymbol{v}_{\mathsf{aux}} := 0$).

5. One defines P_S, $P_{C'}$ and P_v respectively as $\boldsymbol{S} \cdot X + \boldsymbol{S}_{\mathsf{base}}$, $\boldsymbol{C'} \cdot X + \boldsymbol{C'}_{\mathsf{base}}$ and $\boldsymbol{v} \cdot X + \boldsymbol{v}_{\mathsf{base}}$.

Then, to open the evaluations $\boldsymbol{S}_{\mathsf{eval}} := P_S(\phi(i^*))$, $\boldsymbol{C'}_{\mathsf{eval}} := P_{C'}(\phi(i^*))$ and $\boldsymbol{v}_{\mathsf{eval}} := P_v(\phi(i^*))$, the prover just reveals all the seeds $\{\mathsf{seed}_i\}_i$ except seed_{i^*} and the verifier will be able to compute $\boldsymbol{S}_{\mathsf{eval}}$, $\boldsymbol{C'}_{\mathsf{eval}}$ and $\boldsymbol{v}_{\mathsf{eval}}$ as

$$\boldsymbol{S}_{\mathsf{eval}} = \phi(i^*) \cdot \boldsymbol{S}_{\mathsf{aux}} + \sum_{i=1, i \neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot \boldsymbol{S}_{\mathsf{rnd},i}$$

$$\boldsymbol{C'}_{\mathsf{eval}} = \phi(i^*) \cdot \boldsymbol{C'}_{\mathsf{aux}} + \sum_{i=1, i \neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot \boldsymbol{C'}_{\mathsf{rnd},i}$$

$$\boldsymbol{v}_{\mathsf{eval}} = \sum_{i=1, i \neq i^*}^{N} (\phi(i^*) - \phi(i)) \cdot \boldsymbol{v}_{\mathsf{rnd},i}$$

with $(\boldsymbol{S}_{\mathsf{rnd},i}, \boldsymbol{C'}_{\mathsf{rnd},i}, \boldsymbol{v}_{\mathsf{rnd},i}) := \mathsf{PRG}(\mathsf{seed}_i)$.

One-Tree Optimization. Here above, we described how to commit the polynomials P_S, $P_{C'}$ and P_v such that the prover can later open one evaluation. In practice, we perform this commitment phase τ times in parallel. It implies that we use τ all-but-one vector commitments, where each of them uses of GGM tree of N leaves. Therefore, we can optimize those all-but-one vector commitments by using a so-called batched all-but-one vector commitment (BAVC) scheme, which aims to be more efficient than τ independent AVC schemes. In Mirath, we use the BAVC scheme described in [8], which proposes the "one-tree" optimization. Instead of considering τ independent GGM trees of N leaves in parallel, the authors propose to rely on a unique large GGM tree of $\tau \cdot N$ leaves where the i^{th} seed of the e^{th} parallel repetition is associated to the $(e \cdot N + i)^{\text{th}}$ leaf of the large GGM tree. As explained in [8], "opening all but τ leaves of the big tree is more efficient than opening all but one leaf in each of the τ smaller trees, because with high probability some of the active paths in the tree will merge relatively close to the leaves, which reduces the number of internal nodes that need to be revealed." Moreover, the authors of [8] further propose to improve the previous approach using the principle of *grinding*. When the BAVC opening is such that the number of revealed nodes in the revealed sibling paths exceeds a chosen threshold T_{open}, the opening is considered as a failure (*i.e.* it returns $\perp$), forcing the prover/signer recomputing another opening challenge by hashing with an incremented counter. This process is done until the number of revealed nodes is less than T_{open}. For example, if we consider $N = 256$ and $\tau = 16$, the number of revealed nodes is smaller than (or equal to) $T_{\text{open}} := 110$ with probability ≈ 0.2. The selected value of T_{open} induces a rejection probability $p_{\text{rej}} = 1 - 1/\theta$, for some $\theta \in (0, \infty)$, and the signer hence needs to perform an average of θ hash computations for the opening challenge (instead of 1). While this strategy decreases the challenge space by a factor θ, it does not change the average number of hashes that must be computed to succeed an forgery attack

against the signature scheme (since the latter is multiplied by θ). As noticed by the authors of [8], this strategy can be thought of as loosing $\log_2 \theta$ bit of security (because of a smaller challenge space) which are regained thanks to a proof-of-work (performing an average of θ hash computations before getting a valid challenge).

PIOP Computation. We now describe in more details the computation performed over the committed polynomials, namely the computation of P_α for the prover and the computation of $P_\alpha(r)$ for the verifier.

Prover's Computation. We detail here after the computation of the prover to build the polynomial P_E and the polynomial $P_\alpha(X)$. Let us denote $P_E := \boldsymbol{E} \cdot X^2 + \boldsymbol{E}_{\mathsf{mid}} \cdot X + \boldsymbol{E}_{\mathsf{base}}$ and $P_\alpha := \boldsymbol{\alpha} \cdot X^2 + \boldsymbol{\alpha}_{\mathsf{mid}} \cdot X + \boldsymbol{\alpha}_{\mathsf{base}}$. The prover should compute $P_E(X) := P_S(X) \cdot [P_{I_r}(X) \parallel P_{C'}]$, meaning that he should compute

$$\begin{aligned}
\boldsymbol{E} &:= [\boldsymbol{S} \parallel \boldsymbol{S} \cdot \boldsymbol{C}'] \\
\boldsymbol{E}_{\mathsf{mid}} &:= [\boldsymbol{S}_{\mathsf{base}} \parallel \boldsymbol{S}_{\mathsf{base}} \cdot \boldsymbol{C}' + \boldsymbol{S} \cdot \boldsymbol{C}'_{\mathsf{base}}] \\
\boldsymbol{E}_{\mathsf{base}} &:= [\boldsymbol{0}_{m \times r} \parallel \boldsymbol{S}_{\mathsf{base}} \cdot \boldsymbol{C}'_{\mathsf{base}}].
\end{aligned}$$

Then, the prover should compute the polynomial

$$P_\alpha(X) := P_v(X) + \boldsymbol{\Gamma} \cdot \left((\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(P_E(X)) - \boldsymbol{y} \cdot X^2 \right),$$

meaning that he should compute

$$\begin{aligned}
\boldsymbol{\alpha} &:= \boldsymbol{\Gamma} \cdot ((\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(\boldsymbol{E}) - \boldsymbol{y}) \\
\boldsymbol{\alpha}_{\mathsf{mid}} &:= \boldsymbol{\Gamma} \cdot (\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{mid}}) + \boldsymbol{v} \\
\boldsymbol{\alpha}_{\mathsf{base}} &:= \boldsymbol{\Gamma} \cdot (\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{base}}) + \boldsymbol{v}_{\mathsf{base}}.
\end{aligned}$$

Let us remark that, by design, α is always zero so the prover does not need to compute it (by design, the polynomial P_α is of degree at most 1). Therefore, the prover does not need to compute $\boldsymbol{E}$: he just need to compute $\boldsymbol{E}_{\mathsf{mid}}$ and $\boldsymbol{E}_{\mathsf{base}}$ to build $P_\alpha(X) := \boldsymbol{\alpha}_{\mathsf{mid}} \cdot X + \boldsymbol{\alpha}_{\mathsf{base}}$.

Verifier's Computation. We detail here after the computation of the verifier to build the evaluations $\boldsymbol{E}_{\mathsf{eval}} := P_E(r)$ and $\boldsymbol{\alpha}_{\mathsf{eval}} := P_\alpha(r)$. The verifier should compute

$$\begin{aligned}
\boldsymbol{E}_{\mathsf{eval}} &:= P_E(r) \\
&= [P_S(r) P_{I_r}(r) \parallel P_S(r) P_{C'}(r)] \\
&= [\boldsymbol{S}_{\mathsf{eval}} \cdot r \parallel \boldsymbol{S}_{\mathsf{eval}} \cdot \boldsymbol{C}'_{\mathsf{eval}}] \ ,
\end{aligned}$$

together with

$$\begin{aligned}
\boldsymbol{\alpha}_{\mathsf{eval}} &:= P_\alpha(r) \\
&= P_v(r) + \boldsymbol{\Gamma} \cdot \left((\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(P_E(r)) - \boldsymbol{y} \cdot r^2 \right) \\
&= \boldsymbol{v}_{\mathsf{eval}} + \boldsymbol{\Gamma} \cdot \left((\boldsymbol{I}_{m \cdot n - k} \parallel \boldsymbol{H}') \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{eval}}) - \boldsymbol{y} \cdot r^2 \right) \ .
\end{aligned}$$

Fiat-Shamir Transformation & Grinding. To obtain the Mirath signature scheme from the Mirath protocol, we rely on the Fiat-Shamir transformation [26] to remove the prover-verifier interactions. Each verifier's challenge is computed as the output of an extendable-output function (XOF) which takes as input the data that the prover would send before receiving that challenge in an interactive protocol. The Mirath protocol is a 5-round proof system, so there are two challenges: the randomness Γ involved in the definition of P_α, and the evaluation points onto which all the polynomials are evaluated. Since the signature scheme is the non-interactive variant of a 5-round protocol repeated τ times in parallel, the scheme is affected by the attack of Kales and Zaverucha [32]. To avoid incrementing the number τ of parallel protocol repetitions (to have a secure scheme), we draw the first challenge from an exponentially-large set. Therefore, this challenge might be the same across all the parallel repetitions. Using this strategy, to have a secure scheme, $q^{-\mu\cdot\rho}$ and $(2/N)^\tau$ should be negligible. In Mirath scheme, we also use a (explicit) proof-of-work to the Fiat-Shamir hash computation of the last challenge, as proposed in [8]. Together with the opening challenge, the signer samples a w-bit value v_{grinding} and keeps the opening challenge only if this additional value is zero, with w a parameter of the scheme. If this additional value is not zero, then the signer increments a counter and recompute an other opening challenge with an other w-bit value, and he repeats the process until the grinding value is zero. Let us remark that we can use the same counter for this grinding process and the grinding process due to the fact that the [8]'s BAVC scheme might return $\perp$ when the number of revealed nodes is larger than the chosen threshold T_{open}. This strategy increases the cost of hashing the last challenge by a factor 2^w and hence increases the security of w bits. This thus allows to take smaller parameters (N, τ) for the large tree, namely parameters achieving $\lambda - w$ bits of security instead of λ. More precisely, the parameters N, τ and w will be chosen such that $(2/N)^\tau \cdot 2^{-w} \leq 2^{-\lambda}$ to achieve a λ-bit security.

We describe the resulting signature scheme in Algorithms 1 and 2. We added a random salt to have a domain separation between signatures. Let us remark that Mirath uses standard techniques to optimize the signature size: instead of including all the prover's sent data, the signature only contains minimal information that enables the verifier to recompute the prover's sent data and will check if the Fiat-Shamir hashes are consistent with the recomputed data. More precisely,

- Instead of including the evaluations $\boldsymbol{S}_{\mathsf{eval}}$, $\boldsymbol{C}'_{\mathsf{eval}}$ and $\boldsymbol{v}_{\mathsf{eval}}$, the signature includes information (data enabling to derive $\{\mathsf{seed}_i\}_{i\neq i^*}$ and the auxiliary value $(\boldsymbol{S}_{\mathsf{aux}}, \boldsymbol{C}'_{\mathsf{aux}})$) that enables the verifier to derive them.
- Instead of including the polynomial $P_\alpha := \boldsymbol{\alpha}_{\mathsf{mid}} \cdot X + \boldsymbol{\alpha}_{\mathsf{base}}$, the signature only contains $\boldsymbol{\alpha}_{\mathsf{mid}}$. Then, using $\boldsymbol{\alpha}_{\mathsf{eval}} := P_\alpha(\phi(i^*))$ and $\boldsymbol{\alpha}_{\mathsf{mid}}$, the verifier will be able to recompute $\boldsymbol{\alpha}_{\mathsf{base}}$.

Public key: A syndrome MinRank instance $(\boldsymbol{H}, \boldsymbol{y})$.

Secret key: Two matrices $\boldsymbol{S} \in \mathbb{F}_q^{m \times r}$ and $\boldsymbol{C}' \in \mathbb{F}_q^{r \times r(n-r)}$ satisfying $\boldsymbol{H}\,\mathrm{vec}(\boldsymbol{S} \cdot [I_r \parallel \boldsymbol{C}']) = \boldsymbol{y}$.

Step 0: Initialization

1. Uncompress the public and secret keys (if they are in a compressed form).
2. Sample a random $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$.

Step 1: Build & Commit to Witness Polynomials

3. Using a (salted) BAVC, derive τ sets of N seeds $\{\mathsf{seed}_1^{(e)}, \ldots, \mathsf{seed}_N^{(e)}\}_e$ with their commitment digest h_{com}.
4. For each iteration $e \in [1, \ldots, \tau]$:
   (a) For all $i \in [1, \ldots, N]$, expand each seed $\mathsf{seed}_i^{(e)}$ as $(\boldsymbol{S}_{\mathsf{rnd},i}^{(e)}, \boldsymbol{C'}_{\mathsf{rnd},i}^{(e)}, \boldsymbol{v}_{\mathsf{rnd},i}^{(e)})$.
   (b) Compute $\boldsymbol{S}_{\mathsf{acc}}^{(e)} = \sum_{i=1}^{N} \boldsymbol{S}_{\mathsf{rnd},i}^{(e)}$, $\boldsymbol{C'}_{\mathsf{acc}}^{(e)} = \sum_{i=1}^{N} \boldsymbol{C'}_{\mathsf{rnd},i}^{(e)}$, and $\boldsymbol{v}_{\mathsf{acc}}^{(e)} = \sum_{i=1}^{N} \boldsymbol{v}_{\mathsf{rnd},i}^{(e)}$.
   (c) Compute $\boldsymbol{S}_{\mathsf{base}}^{(e)} = -\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{S}_{\mathsf{rnd},i}^{(e)}$, $\boldsymbol{C'}_{\mathsf{base}}^{(e)} = -\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{C'}_{\mathsf{rnd},i}^{(e)}$, and $\boldsymbol{v}_{\mathsf{base}}^{(e)} = -\sum_{i=1}^{N} \phi(i) \cdot \boldsymbol{v}_{\mathsf{rnd},i}^{(e)}$.
   (d) Compute $\boldsymbol{S}_{\mathsf{aux}}^{(e)} = \boldsymbol{S} - \boldsymbol{S}_{\mathsf{acc}}^{(e)}$ and $\boldsymbol{C'}_{\mathsf{aux}}^{(e)} = \boldsymbol{C'} - \boldsymbol{C'}_{\mathsf{acc}}^{(e)}$, and set $\boldsymbol{v}^{(e)}$ as $\boldsymbol{v}_{\mathsf{acc}}^{(e)}$.
5. Compute $h_{\mathrm{sh}} = \mathsf{Hash}_1(h_{\mathsf{com}}, (\boldsymbol{S}_{\mathsf{aux}}^{(e)}, \boldsymbol{C'}_{\mathsf{aux}}^{(e)})_{e \in [1,\ldots,\tau]})$

Step 2: Compute the polynomial proof $P_\alpha(X)$

6. Sample $\boldsymbol{\Gamma} \xleftarrow{\$} \mathsf{PRG}(h_{\mathrm{sh}})$ where $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times mn-k}$.
7. For each iteration $e \in [1, \ldots, \tau]$:
   (a) Compute $P_E^{(e)}(X)$ by computing

$$\boldsymbol{E}_{\mathsf{mid}}^{(e)} := [\boldsymbol{S}_{\mathsf{base}}^{(e)} \parallel \boldsymbol{S}_{\mathsf{base}}^{(e)} \cdot \boldsymbol{C'}^{(e)} + \boldsymbol{S}^{(e)} \cdot \boldsymbol{C'}_{\mathsf{base}}^{(e)}],$$
$$\boldsymbol{E}_{\mathsf{base}}^{(e)} := [\boldsymbol{0}_{m \times r} \parallel \boldsymbol{S}_{\mathsf{base}}^{(e)} \cdot \boldsymbol{C'}_{\mathsf{base}}^{(e)}].$$

   (b) Compute the polynomial $P_\alpha^{(e)} := \boldsymbol{\alpha}_{\mathsf{mid}}^{(e)} \cdot X + \boldsymbol{\alpha}_{\mathsf{base}}^{(e)}$ by computing

$$\boldsymbol{\alpha}_{\mathsf{mid}}^{(e)} := \boldsymbol{\Gamma} \cdot (\boldsymbol{I}_{m \cdot n-k} \parallel \boldsymbol{H'}) \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{mid}}^{(e)}) + \boldsymbol{v}^{(e)},$$
$$\boldsymbol{\alpha}_{\mathsf{base}}^{(e)} := \boldsymbol{\Gamma} \cdot (\boldsymbol{I}_{m \cdot n-k} \parallel \boldsymbol{H'}) \cdot \mathrm{vec}(\boldsymbol{E}_{\mathsf{base}}^{(e)}) + \boldsymbol{v}_{\mathsf{base}}^{(e)}.$$

8. Compute $h_{\mathrm{piop}} = \mathsf{Hash}_2\left(\mathsf{pk}, \mathsf{salt}, \mathsf{msg}, h_{\mathrm{sh}}, (\boldsymbol{\alpha}_{\mathsf{mid}}^{(e)}, \boldsymbol{\alpha}_{\mathsf{base}}^{(e)})_{e \in [1,\ldots,\tau]}\right)$.

Step 3: Open Random Evaluations

9. Set $\mathsf{ctr} := 0$.
10. Sample $\left(\{i^{*(e)}\}_e, v_{\mathrm{grinding}}\right) \xleftarrow{\$} \mathsf{PRG}(h_{\mathrm{piop}}, \mathrm{ctr})$ where $i^{*(e)} \in [1, \ldots, N]$ for all $e \in [1, \ldots, \tau]$ and $v_{\mathrm{grinding}} \in \{0,1\}^w$.
11. Compute the BAVC's opening proof π_{BAVC} for $\{\mathsf{seed}_i^e\}_{e, i \neq i^{*(e)}}$.
12. If $v_{\mathrm{grinding}} \neq 0$ or $\pi_{\mathsf{BAVC}} = \perp$, increment ctr and go to Step 10.
13. Output the signature $\sigma = \left(\mathsf{salt} \mid \mathsf{ctr} \mid h_{\mathrm{piop}} \mid \pi_{\mathsf{BAVC}} \mid \left(\boldsymbol{S}_{\mathsf{aux}}^{(e)}, \boldsymbol{C'}_{\mathsf{aux}}^{(e)}, \boldsymbol{\alpha}_{\mathsf{mid}}^{(e)}\right)_{e \in [1,\ldots,\tau]}\right)$.

Algorithm 1. High level description of Mirath Sign algorithm

Public key: A syndrome MinRank instance $(\boldsymbol{H}, \boldsymbol{y})$.

Step 0: Initialization

1. Uncompress the public key (if it is in a compressed form).
2. Parse the signature as $\left(\mathsf{salt} \mid \mathsf{ctr} \mid h_{\mathrm{piop}} \mid \pi_{\mathsf{BAVC}} \mid \left(\boldsymbol{S}^{(e)}_{\mathsf{aux}}, \boldsymbol{C'}^{(e)}_{\mathsf{aux}}, \boldsymbol{\alpha}^{(e)}_{\mathsf{mid}}\right)_{e\in[1,\dots,\tau]}\right)$.

Step 1: Computing Opened Evaluations

3. Sample $\left(\{i^{*(e)}\}_e, v_{\mathrm{grinding}}\right) \xleftarrow{\$} \mathsf{PRG}(h_{\mathrm{piop}}, \mathrm{ctr})$ where $i^{*(e)} \in [1,\dots,N]$ for all $e \in [1,\dots,\tau]$ and $v_{\mathrm{grinding}} \in \{0,1\}^w$.
4. Using π_{BAVC} and $\{i^{*(e)}\}_e$, recover $\{\mathsf{seed}^e_i\}_{e,i\neq i^{*(e)}}$ with the reconstruction algorithm of the BAVC scheme, together with the commitment digest h_{com}.
5. For each iteration $e \in [1,\dots,\tau]$:
   (a) For all $i \in [1,\dots,N]\backslash\{i^{*(e)}\}$, expand each seed $\mathsf{seed}^{(e)}_i$ as $(\boldsymbol{S}^{(e)}_{\mathsf{rnd},i}, \boldsymbol{C'}^{(e)}_{\mathsf{rnd},i}, \boldsymbol{v}^{(e)}_{\mathsf{rnd},i})$.
   (b) Compute

$$\boldsymbol{S}^{(e)}_{\mathsf{eval}} = \phi(i^{*(e)}) \cdot \boldsymbol{S}^{(e)}_{\mathsf{aux}} + \sum_{i=1, i\neq i^{*(e)}}^{N} (\phi(i^{*(e)}) - \phi(i)) \cdot \boldsymbol{S}^{(e)}_{\mathsf{rnd},i}$$

$$\boldsymbol{C'}^{(e)}_{\mathsf{eval}} = \phi(i^{*(e)}) \cdot \boldsymbol{C'}^{(e)}_{\mathsf{aux}} + \sum_{i=1, i\neq i^{*(e)}}^{N} (\phi(i^{*(e)}) - \phi(i)) \cdot \boldsymbol{C'}^{(e)}_{\mathsf{rnd},i}$$

$$\boldsymbol{v}^{(e)}_{\mathsf{eval}} = \sum_{i=1, i\neq i^{*(e)}}^{N} (\phi(i^{*(e)}) - \phi(i)) \cdot \boldsymbol{v}^{(e)}_{\mathsf{rnd},i} \ .$$

6. Compute $h_{\mathrm{sh}} = \mathsf{Hash}_1(h_{\mathsf{com}}, (\boldsymbol{S}^{(e)}_{\mathsf{aux}}, \boldsymbol{C'}^{(e)}_{\mathsf{aux}})_{e\in[1,\dots,\tau]})$

Step 2: Recompute the polynomial proof $P_\alpha(X)$

7. Sample $\boldsymbol{\Gamma} \xleftarrow{\$} \mathsf{PRG}(h_{\mathrm{sh}})$ where $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho\times mn-k}$.
8. For each iteration $e \in [1,\dots,\tau]$:
   (a) Compute the evaluation $\boldsymbol{E}^{(e)}_{\mathsf{eval}} := P^{(e)}_E(\phi(i^{*(e)}))$ as

$$\boldsymbol{E}^{(e)}_{\mathsf{eval}} = [\boldsymbol{S}^{(e)}_{\mathsf{eval}} \cdot \phi(i^{*(e)}) \parallel \boldsymbol{S}^{(e)}_{\mathsf{eval}} \cdot \boldsymbol{C'}^{(e)}_{\mathsf{eval}}].$$

   (b) Compute the evaluation $\boldsymbol{\alpha}^{(e)}_{\mathsf{eval}} := P^{(e)}_\alpha(\phi(i^{*(e)}))$ as

$$\boldsymbol{\alpha}^{(e)}_{\mathsf{eval}} = \boldsymbol{v}_{\mathsf{eval}} + \boldsymbol{\Gamma} \cdot \left((\boldsymbol{I}_{m\cdot n-k} \parallel \boldsymbol{H'}) \cdot \mathrm{vec}(\boldsymbol{E}^{(e)}_{\mathsf{eval}}) - \boldsymbol{y} \cdot \phi(i^{*(e)})^2\right).$$

   (c) Deduce $\boldsymbol{\alpha}^{(e)}_{\mathsf{base}}$ as

$$\boldsymbol{\alpha}^{(e)}_{\mathsf{base}} = \boldsymbol{\alpha}^{(e)}_{\mathsf{eval}} - \boldsymbol{\alpha}^{(e)}_{\mathsf{mid}} \cdot \phi(i^{*(e)}).$$

9. Compute $h'_{\mathrm{piop}} = \mathsf{Hash}_2\left(\mathsf{pk}, \mathsf{salt}, \mathsf{msg}, h_{\mathrm{sh}}, (\boldsymbol{\alpha}^{(e)}_{\mathsf{mid}}, \boldsymbol{\alpha}^{(e)}_{\mathsf{base}})_{e\in[1,\dots,\tau]}\right)$.

Step 3: Verification

10. Check that $h'_{\mathrm{piop}} =_? h_{\mathrm{piop}}$ and $v_{\mathrm{grinding}} = 0$.

Algorithm 2. High level description of Mirath Verify algorithm

4 Detailed Description of Mirath

4.1 Notations

We let $y \leftarrow \mathsf{A}(x)$ denote the operation that runs algorithm A on input x and assigns the output to the variable y. The notation $a \mathrel{+}= b$ means that to the variable a it is assigned the value $a + b$, where the addition is performed component-wise if a and b are tuples. We employ arrays, and we let $\mathsf{a}[i]$ denote the ith element of the array a. The main variable names employed in the algorithms are collected in Table 2.

Vectors and Matrices:		
$\boldsymbol{E}$	$\mathbb{F}_q^{m \times n}$	Matrix solution of the Dual Support Modeling.
$\boldsymbol{S}, \boldsymbol{C'}$	$\mathbb{F}_q^{m \times r}, \mathbb{F}_q^{r \times (n-r)}$	Matrices of the secret key: $\boldsymbol{E} = \boldsymbol{S}[\boldsymbol{I}_r \ \boldsymbol{C'}]$.
$\boldsymbol{H'}$	$\mathbb{F}_q^{(m \cdot n-k) \times k}$	Matrix of the public key.
$\boldsymbol{y}$	$\mathbb{F}_q^{(mn-k) \times k}$	Vector of the public key: $[\boldsymbol{I}_{mn-k} \boldsymbol{H'}] \operatorname{vec}(\boldsymbol{E}) = \boldsymbol{y}$.
$\boldsymbol{v}$	$\mathbb{F}_{q^\mu}^{\rho \times 1}$	Random masking vector.
$\boldsymbol{\Gamma}$	$\mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$	Challenge matrix.
$\phi(i^*)$	$\mathbb{F}_{q^\mu}$	Evaluation points from the second challenge $\{i^*[e]\}_{e<\tau}$.
Seeds and Salt:		
$\mathsf{seed_{pk}}$	$\{0,1\}^\lambda$	Seed for the generation of the matrix $\boldsymbol{H'}$ of the public key.
$\mathsf{seed_{sk}}$	$\{0,1\}^\lambda$	Seed for the generation of the matrices $\boldsymbol{S}, \boldsymbol{C'}$ of the secret key.
tree	$\{0,1\}^{2\lambda\tau N-1}$	Binary tree of seeds.
rseed	$\{0,1\}^\lambda$	Root seed of tree.
seeds	$\{0,1\}^{\lambda\tau N}$	Leaf seeds of tree.
salt	$\{0,1\}^\lambda$	Salt for domain separation between signatures.
Other Variables:		
msg	$\{0,1\}^*$	Message to be be signed.
commit	$\{0,1\}^{2\lambda\tau N}$	Commitments of seeds.
key	$\{0,1\}^{4\lambda\tau N-1}$	tree $\|\|$ commit.
path	$\{0,1\}^{\lambda \cdot T_{\text{open}}}$	Opening of the BAVC leaves in tree.
commit_{i^*}	$\{0,1\}^{2\lambda\tau}$	Commitments of the unopened BAVC leaves in tree.
π_{BAVC}	$\{0,1\}^{\lambda(T_{\text{open}}+2\tau)}$	path $\|\|$ commit_{i^*}.
h_{com}	$\{0,1\}^{2\lambda}$	Seed commitment (hash digest of commit).
h_{sh}	$\{0,1\}^{2\lambda}$	Commitment of the witness polynomials.
h_{piop}	$\{0,1\}^{2\lambda}$	Polynomial proof commitment.
ctr	$\{0,1\}^{64}$	Counter for grinding and proof-of-work optimizations.
σ	$\{0,1\}^*$	Signature output by the signing algorithm.

Table 2. Algorithmic notation.

4.2 Finite Fields Representation

Elements in $\mathbb{F}_q$. Mirath proposes parameters for our scheme over the fields $\mathbb{F}_2$ and $\mathbb{F}_{16}$. Every element in $\mathbb{F}_{16}$ is represented by a degree-3 binary polynomial $a_3x^3+a_2x^2+a_1x+a_0$. An element $a_3x^3+a_2x^2+a_1x+a_0 \in \mathbb{F}_{16}$ is stored as the four-bit integer $a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2 + a_0$. The multiplication of two elements is implemented as a polynomial multiplication performed modulo $f(x) = x^4+x+1$.

Elements in $\mathbb{F}_{q^\mu}$. Every element in $\mathbb{F}_{q^\mu}$ is represented by a degree-$(\mu-1)$ polynomial $a_{\mu-1}x^{\mu-1} + \cdots + a_1x + a_0$, where $a_0, \ldots, a_{\mu-1} \in \mathbb{F}_q$. We store an element $a_{\mu-1}x^{\mu-1}+\cdots+a_1x+a_0 \in \mathbb{F}_{q^\mu}$ as the integer with binary representation

$$\mathtt{bin}(a_{\mu-1}) \parallel \mathtt{bin}(a_{\mu-2}) \parallel \cdots \parallel \mathtt{bin}(a_1) \parallel \mathtt{bin}(a_0),$$

where $\mathtt{bin}(a_i)$ is the binary representation of $a_i \in \mathbb{F}_q$. We use $(\mu \log q)$ bits to store a single $\mathbb{F}_{q^\mu}$-element. The multiplication of two elements in $\mathbb{F}_{q^\mu}$ is implemented as a polynomial multiplication modulo an irreducible polynomial $f(x)$, where $f(x)$ is chosen as given in Table 3.

q	μ	$f(x)$
2	8	x^8+x^4+x+1
2	11	$x^{11}+x^2+1$
2	12	$x^{12}+x^3+1$
16	2	x^2+x+1
16	3	x^3+x+1

Table 3. Polynomial modulus $f(x)$ for multiplications in $\mathbb{F}_{q^\mu}$.

4.3 Matrices Representation

The entries of a matrix are stored in column-major order using byte arrays. In this subsection, M_j denotes the j-th column of a matrix M. In addition, $p = 8/\log q$ defines the maximum number of $\mathbb{F}_q$ elements that can be stored in one byte.

Matrices over $\mathbb{F}_q$. An matrix $M \in \mathbb{F}_q^{n_r \times n_c}$ is represented as a byte-array $\mathtt{M}$ of length $\mathsf{nb}_c \cdot n_c$, where $\mathsf{nb}_c = \lceil n_r \log q/8 \rceil$. This array is represented as $\mathtt{M} = \mathtt{M}_0 \parallel \mathtt{M}_1 \parallel \cdots \parallel \mathtt{M}_{n_c-1}$, where each $\mathtt{M}_j$ is a byte-array of length nb_c storing the column M_j as it is explained below.

First suppose

$$M_j = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n_r-1} \end{bmatrix} \in \mathbb{F}_q^{n_r}.$$

Given $0 \leq i \leq \mathsf{nb}_c - 2$. The i-th byte $\mathtt{M}_j\,\mathtt{[}i\mathtt{]}$ in the byte array $\mathtt{M}_j$ stores the p entries $a_{p\cdot i}, a_{p\cdot i+1}, \ldots, a_{p\cdot i+p-1}$ by defining

$$\mathtt{M}_j\,\mathtt{[}i\mathtt{]} = \mathtt{bin}(a_{p\cdot i+p-1}) \parallel \mathtt{bin}(a_{p\cdot i+p-2}) \parallel \cdots \parallel \mathtt{bin}(a_{p\cdot i+1}) \parallel \mathtt{bin}(a_{p\cdot i}),$$

where $\mathtt{bin}(a)$ is the binary representation of $a \in \mathbb{F}_q$. The last byte $\mathtt{M}_j\,\mathtt{[nb}_c - 1\mathtt{]}$ in $\mathtt{M}_j$ stores the last $p' = n_r \bmod p$ entries $a_{n_r-1-p'}, \ldots, a_{n_r-1}$ of M_j, that is,

$$\mathtt{M}_j\,\mathtt{[nb}_c - 1\mathtt{]} = \mathbf{0} \parallel \mathtt{bin}(a_{p(\mathtt{nb}_c-1)+p'-1}) \parallel \cdots \parallel \mathtt{bin}(a_{p(\mathtt{nb}_c-1)+1}) \parallel \mathtt{bin}(a_{p(\mathtt{nb}_c-1)}),$$

where $\mathbf{0}$ denotes a bit array of $(8 - p' \cdot \log q)$ zeros.

Matrices over $\mathbb{F}_q$ from byte arrays. Algorithm 1 describes how to transform a byte array $\mathtt{A}$ into a matrix in $\mathbb{F}_q^{n_r \times n_c}$.

Algorithm 1 Transform a byte array into a matrix in $\mathbb{F}^{n_r \times n_c}$.

SetToMatrix($\mathtt{A}, \mathbb{F}^{n_r \times n_c}$)

1 : $p \leftarrow 8/\log q$ ▷ Maximun number of elements stored in one byte.
2 : $\mathsf{mask} \leftarrow 255$
3 : **if** $\mathbb{F} = \mathbb{F}_q$ **then**
4 : $\quad \mathsf{nb}_c \leftarrow \lceil n_r/p \rceil$ ▷ Number of bytes used to store one column.
5 : $\quad a \leftarrow n_r \bmod p$
6 : $\quad$ **if** $a > 0$ **then**
7 : $\quad\quad \mathsf{mask} \leftarrow \lfloor \mathsf{mask}/2^{(p-a)\cdot \log q} \rfloor$
8 : $\quad$ **for** j from 1 to n_c **do**
9 : $\quad\quad \mathtt{A}[j \cdot \mathsf{nb}_c - 1] \leftarrow \mathtt{A}[j \cdot \mathsf{nb}_c - 1] \otimes \mathsf{mask}$ ▷ $\otimes$ means bitwise multiplication
10 : **if** $\mathbb{F} = \mathbb{F}_{q^u}$ **then**
11 : $\quad \mathsf{nb}_e \leftarrow \lceil \mu/p \rceil$ ▷ Number of bytes used to store one $\mathbb{F}_{q^\mu}$ element.
12 : $\quad a \leftarrow \mu \bmod p$
13 : $\quad$ **if** $a > 0$ **then**
14 : $\quad\quad \mathsf{mask} \leftarrow \lfloor \mathsf{mask}/2^{(p-a)\cdot \log q} \rfloor$
15 : $\quad$ **for** j from 0 to $n_c n_r - 1$ **do**
16 : $\quad\quad \mathtt{A}[j \cdot \mathsf{nb}_e] \leftarrow \mathtt{A}[j \cdot \mathsf{nb}_e] \otimes \mathsf{mask}$ ▷ $\otimes$ means bitwise multiplication
17 : **return** $\mathtt{A}$

Matrices over $\mathbb{F}_{q^\mu}$. An element $M \in \mathbb{F}_{q^\mu}^{n_r \times n_c}$ is represented as a byte array $\mathtt{M}$ of length $\mathsf{nb}_c \cdot n_c$, where $\mathsf{nb}_c = \lceil \mu \log q/8 \rceil n_r$. The (i, j) entry of M is stored in the $s = \lceil \mu \log q/8 \rceil$ bytes $\mathtt{M[nb}_c \cdot j + s \cdot i], \ldots, \mathtt{M[nb}_c \cdot j + s \cdot i + s - 1\mathtt{]}$ as explained in Section 4.2.

Parsing of matrices over $\mathbb{F}_q$ and $\mathbb{F}_{q^\mu}$ from byte arrays. Algorithm 2 describes how we pack in Mirath a matrix defined over $\mathbb{F}_q$ or $\mathbb{F}_{q^\mu}$ into a byte array. Algorithm 3 gives the way the matrix is unpacked back. In the description of these two algorithms, we assume that the copying from the matrix or the bytes array is done bit by bit. Algorithm 2 is used in UnparseSignature and Algorithm 3 is used in ParseSignature.

Algorithm 2 Unparse matrix.

UnparseMatrix($\mathtt{A}, \boldsymbol{M}, \mathtt{i}_b, \mathbb{F}^{n_r \times n_c}$)

if $\mathbb{F} = \mathbb{F}_q$ **then**
  $\mathsf{nb}_c \leftarrow 8\lceil n_r \log q/8 \rceil$
  for j from 0 to $(n_c - 1)$ **do**
    $\mathtt{A}[\mathtt{i}_b, \mathtt{i}_b + n_r \log q - 1] \leftarrow \boldsymbol{M}[j \cdot \mathsf{nb}_c, j \cdot \mathsf{nb}_c + n_r \log q - 1]$ ▷ Bit-to-bit copy
    $\mathtt{i}_b$ += $n_r \log q$
if $\mathbb{F} = \mathbb{F}_{q^\mu}$ **then**
  $\mathsf{nb}_r \leftarrow 8\lceil \log q^\mu/8 \rceil$
  $\mathsf{nb}_c \leftarrow \mathsf{nb}_r \cdot n_r$
  for j from 0 to $(n_c - 1)$ **do**
    for i from 0 to $(n_r - 1)$ **do**
      $\mathtt{A}[\mathtt{i}_b, \mathtt{i}_b + \log q^\mu - 1] \leftarrow \boldsymbol{M}[j \cdot \mathsf{nb}_c + i \cdot \mathsf{nb}_r, j \cdot \mathsf{nb}_c + i \cdot \mathsf{nb}_r + \log q^\mu - 1]$
      $\mathtt{i}_b$ += $\log q^\mu$
return $\mathtt{A}, \mathtt{i}_b$

4.4 Hash Functions and Commitments

Hash Functions. Several routines in Mirath involve cryptographic hashing. These routines all use a common cryptographic hash function $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^{2\lambda}$. For the sake of simplicity, we write $\mathsf{Hash}(\mathsf{obj}_1, \mathsf{obj}_2, \dots)$ as a shortcut for the hashing of a binary string obtained by concatenating the binary strings representing the objects $\mathsf{obj}_1, \mathsf{obj}_2, \dots$. These hash functions are implemented using SHA3-λ along with domain separation as follows:

$$\mathsf{Hash}_1(\mathsf{data}) := \mathsf{Hash}(1, \mathsf{data})$$
$$\mathsf{Hash}_2(\mathsf{data}) := \mathsf{Hash}(2, \mathsf{data})$$
$$\mathsf{Hash}_3(\mathsf{data}) := \mathsf{Hash}(3, \mathsf{data})$$

where each of the prefix 1, 2, and 3 is encoded in one byte.

Hash_1 is used in CommitWitnessPolynomials and ComputeEvaluations. Hash_2 is used in Sign and Verify. Hash_3 is used in BAVC.Commit and BAVC.Reconstruct.

Algorithm 3 Parse matrix.

ParseMatrix($\mathtt{A}, \mathtt{i}_b, \mathbb{F}^{n_r \times n_c}$)

1 : **if** $\mathbb{F} = \mathbb{F}_q$ **then**
2 : $\quad \mathsf{nb}_c \leftarrow 8\lceil n_r \log q/8 \rceil$
3 : $\quad \boldsymbol{M} \leftarrow \mathbf{0}_{(\mathsf{nb}_c \cdot n_c)}$ ▷ An array of $(\mathsf{nb}_c \cdot n_c)$ zero bytes
4 : $\quad$ **for** j from 0 to $(n_c - 1)$ **do**
5 : $\quad\quad \boldsymbol{M}[j \cdot \mathsf{nb}_c, j \cdot \mathsf{nb}_c + n_r \log q - 1] \leftarrow \mathtt{A}[\mathtt{i}_b, \mathtt{i}_b + n_r \log q - 1]$ ▷ Bit-to-bit copy
6 : $\quad\quad \mathtt{i}_b$ += $n_r \log q$
7 : **if** $\mathbb{F} = \mathbb{F}_{q^\mu}$ **then**
8 : $\quad \mathsf{nb}_r \leftarrow 8\lceil \log q^\mu/8 \rceil$
9 : $\quad \mathsf{nb}_c \leftarrow \mathsf{nb}_r \cdot n_r$
10 : $\quad \boldsymbol{M} \leftarrow \mathbf{0}_{(\mathsf{nb}_c \cdot n_c)}$ ▷ An array of $(\mathsf{nb}_c \cdot n_c)$ zero bytes
11 : $\quad$ **for** j from 0 to $(n_c - 1)$ **do**
12 : $\quad\quad$ **for** i from 0 to $(n_r - 1)$ **do**
13 : $\quad\quad\quad \boldsymbol{M}[j \cdot \mathsf{nb}_c + i \cdot \mathsf{nb}_r, j \cdot \mathsf{nb}_c + i \cdot \mathsf{nb}_r + \log q^\mu - 1] \leftarrow \mathtt{A}[\mathtt{i}_b, \mathtt{i}_b + \log q^\mu - 1]$
14 : $\quad\quad\quad \mathtt{i}_b$ += $\log q^\mu$
15 : **return** $\boldsymbol{M}, \mathtt{i}_b$

4.5 Randomness Generation and Sampling

Random Objects. TRG.Seed(λ) returns a random binary string of length λ. It is instantiated using the NIST provided randombytes function, which returns $\lceil \lambda/8 \rceil$ bytes from the system entropy. This functionality is notably used to generate the public key and secret key seeds.

Pseudorandom Objects. Several algorithms used in Mirath require the generation of pseudorandom objects. These objects are generated from pseudorandom bytes produced with a PRG based on a block cipher Encrypt-λ or the XOF Shake.

Encrypt-λ(key, msg) output the encryption $c \in \{0,1\}^\lambda$ of the block msg $\in \{0,1\}^\lambda$ using the key key $\in \{0,1\}^\lambda$. The selection of the particular block cipher depends on the security level. In Mirath Encrypt-128 = AES-128 and Encrypt-λ = Rijndael-λ for $\lambda = 192, 256$. This functionality is used in Algorithm 4 and Algorithm 5.

Shake is an extendable-output function [17]. It consists of the initialization function Shake.Init that returns a new instance of the Shake object prg, an absorbing function Shake.Absorb(prg, string) that returns the updated shake instance prg after absorbing the string of bits string, and a squeeze function Shake.Squeeze(prg, nb) that returns the updated shake instance prg and a string

of nb pseudorandom bytes.

Algorithm 4 Children nodes generation.

$\mathsf{ExpandSeed}_a(\mathsf{salt}, \mathsf{seed}, \mathsf{idx})$

1 : $\mathsf{salt}_0 \leftarrow \mathsf{salt}[0, \lambda]$ ▷ First λ bits in salt.
2 : $\mathsf{msg}_0 \leftarrow \mathsf{salt}_0 \oplus (\mathtt{bin}_{\lambda-40}(a) \parallel \mathtt{bin}_{32}(\mathsf{idx}) \parallel \mathtt{bin}_8(0))$
3 : $\mathsf{msg}_1 \leftarrow \mathsf{salt}_0 \oplus (\mathtt{bin}_{\lambda-40}(a) \parallel \mathtt{bin}_{32}(\mathsf{idx}) \parallel \mathtt{bin}_8(1))$
4 : ▷ $\mathtt{bin}_x(i)$ is the x-bit representation of i.
5 : $\mathsf{seed}_l \leftarrow \text{Encrypt-}\lambda(\mathsf{seed}, \mathsf{msg}_0)$
6 : $\mathsf{seed}_r \leftarrow \text{Encrypt-}\lambda(\mathsf{seed}, \mathsf{msg}_1)$
7 : **return** $(\mathsf{seed}_l, \mathsf{seed}_r)$

$\mathsf{ExpandSeed}_a(\mathsf{salt}, \mathsf{seed}, \mathsf{idx})$ returns a pseudorandom element in $\{0,1\}^\lambda \times \{0,1\}^\lambda$ built from a salt $\mathsf{salt} \in \{0,1\}^{128}$, a seed $\mathsf{seed} \in \{0,1\}^\lambda$ and a nonnegative integer $\mathsf{idx} < N$. It is described in Algorithm 4, and used in BAVC.Commit and BAVC.Reconstruct.

Algorithm 5 Pseudorandom element in $\mathbb{F}_q^{m\times r} \times \mathbb{F}_q^{r\times(n-r)} \times \mathbb{F}_{q^\mu}^{\rho\times 1}$.

$\mathsf{ExpandSeedShares}(\mathsf{salt}, \mathsf{seed})$

1 : $\mathsf{salt}_0 \leftarrow \mathsf{salt}[0, \lambda]$ ▷ First λ bits in salt.
2 : $p \leftarrow 8/\log q$
3 : $\mathsf{nb}_s \leftarrow \lceil m/p \rceil r$ ▷ Number of bytes to store $\boldsymbol{S}_{\mathsf{rnd}} \in \mathbb{F}_q^{m\times r}$.
4 : $\mathsf{nb}_c \leftarrow \lceil r/p \rceil \cdot (n-r)$ ▷ Number of bytes to store $\boldsymbol{C}_{\mathsf{rnd}} \in \mathbb{F}_q^{r\times(n-r)}$.
5 : $\mathsf{nb}_v \leftarrow \lceil \mu/p \rceil \rho$ ▷ Number of bytes to store $\boldsymbol{v}_{\mathsf{rnd}} \in \mathbb{F}_{q^\mu}^{\rho\times 1}$.
6 : $\mathsf{nb} \leftarrow \mathsf{nb}_s + \mathsf{nb}_c + \mathsf{nb}_v$
7 : $\mathsf{nenc} \leftarrow \lceil \mathsf{nb} \cdot 8/\lambda \rceil$
8 : $\mathtt{A} \leftarrow \emptyset$
9 : **for** i from 0 to $\mathsf{nenc} - 1$ **do**
10 : $\mathtt{A} \leftarrow \mathtt{A} \parallel \text{Encrypt-}\lambda(\mathsf{seed}, \mathsf{salt}_0 \oplus \mathtt{bin}_\lambda(i))$
11 : ▷ $\mathtt{bin}_\lambda(i)$ is the λ-bit representation of i.
12 : $\boldsymbol{S}_{\mathsf{rnd}} \leftarrow \text{SetToMatrix}((\mathtt{A[0]}, \ldots, \mathtt{A}[\mathsf{nb}_s - 1]), \mathbb{F}_q^{m\times r})$
13 : $\boldsymbol{C}'_{\mathsf{rnd}} \leftarrow \text{SetToMatrix}((\mathtt{A}[\mathsf{nb}_s], \ldots, \mathtt{A}[\mathsf{nb}_s + \mathsf{nb}_c - 1]), \mathbb{F}_q^{r\times(n-r)})$
14 : $\boldsymbol{v}_{\mathsf{rnd}} \leftarrow \text{SetToMatrix}((\mathtt{A}[\mathsf{nb}_s + \mathsf{nb}_c] \parallel \cdots \parallel \mathtt{A}[\mathsf{nb} - 1]), \mathbb{F}_{q^\mu}^{\rho\times 1})$
15 : **return** $(\boldsymbol{S}_{\mathsf{rnd}}, \boldsymbol{C}'_{\mathsf{rnd}}, \boldsymbol{v}_{\mathsf{rnd}})$

ExpandSeedShares(salt, seed) returns a pseudorandom element in $\mathbb{F}_q^{m\times r} \times \mathbb{F}_q^{r\times(n-r)} \times \mathbb{F}_{q^\mu}^{\rho\times 1}$ that is generated from a salt $\mathsf{salt} \in \{0,1\}^{128}$ and a seed $\mathsf{seed} \in \{0,1\}^{\lambda}$. Algorithm 5 describes this routine, which is used during signing and verification within CommitWitnessPolynomials and ComputeEvaluations, respectively.

Algorithm 6 Routine ExpandSeedPublicMatrix.

ExpandSeedPublicMatrix($\mathsf{seed}_{\mathsf{pk}}$)

1: $p \leftarrow 8/\log q$
2: $\mathsf{nb} \leftarrow \lceil (mn-k)/p \rceil k$ ▷ Number of bytes to store $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k)\times k}$.
3: $\mathsf{prg} \leftarrow$ Shake.Init()
4: $\mathsf{prg} \leftarrow$ Shake.Absorb($\mathsf{prg}, \mathsf{seed}_{\mathsf{pk}}$)
5: $\mathtt{H} \leftarrow$ Shake.Squeeze($\mathsf{prg}, \mathsf{nb}$)
6: $\boldsymbol{H}' \leftarrow$ SetToMatrix($\mathtt{H}, \mathbb{F}_q^{(mn-k)\times k}$)
7: **return** $\boldsymbol{H}'$

ExpandSeedPublicMatrix($\mathsf{seed}_{\mathsf{pk}}$) returns the matrix $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k)\times k}$ which is expended from a seed $\mathsf{seed}_{\mathsf{pk}}$. It is described in Algorithm 6, and it is used in KeyGen and Verify within the DecompressPK function.

Algorithm 7 Routine ExpandSeedSecretMatrices.

ExpandSeedSecretMatrices($\mathsf{seed}_{\mathsf{sk}}$)

1: $p \leftarrow 8/\log q$
2: $\mathsf{nb}_s \leftarrow \lceil m/p \rceil r$ ▷ Number of bytes to store $\boldsymbol{S} \in \mathbb{F}_q^{m\times r}$.
3: $\mathsf{nb}_c \leftarrow \lceil r/p \rceil \cdot (n-r)$ ▷ Number of bytes to store $\boldsymbol{C}' \in \mathbb{F}_q^{r\times(n-r)}$.
4: $\mathsf{nb} \leftarrow \mathsf{nb}_s + \mathsf{nb}_c$
5: $\mathsf{prg} \leftarrow$ Shake.Init()
6: $\mathsf{prg} \leftarrow$ Shake.Absorb($\mathsf{prg}, \mathsf{seed}_{\mathsf{sk}}$)
7: $\mathtt{SC} \leftarrow$ Shake.Squeeze($\mathsf{prg}, \mathsf{nb}$)
8: $\boldsymbol{S} \leftarrow$ SetToMatrix($(\mathtt{SC[0]}, \ldots, \mathtt{SC}[\mathsf{nb}_s - 1]), \mathbb{F}_q^{m\times r}$)
9: $\boldsymbol{C}' \leftarrow$ SetToMatrix($(\mathtt{SC}[\mathsf{nb}_s], \ldots, \mathtt{SC}[\mathsf{nb} - 1]), \mathbb{F}_q^{r\times(n-r)}$)
10: **return** $(\boldsymbol{S}, \boldsymbol{C}')$

ExpandSeedSecretMatrices(seed) returns the secret matrices $\boldsymbol{S} \in \mathbb{F}_q^{m\times r}$ and $\boldsymbol{C}' \in \mathbb{F}_q^{r\times(n-r)}$. It is used in KeyGen and Sign within the DecompressSK function that

also returns the matrices $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k)\times k}$ and the public key pk.

Algorithm 8 Expand challenge evaluation points.

ExpandChallengeEvaluationPoints(h, ctr)

1 : nbits $\leftarrow (\tau \cdot \log N) + w$
2 : nb $\leftarrow \lceil \mathsf{nbits}/8 \rceil$
3 : prg $\leftarrow$ Shake.Init()
4 : prg $\leftarrow$ Shake.Absorb(prg, $h \parallel \mathtt{bin}_{64}(\mathsf{ctr})$)
5 : rnd $\leftarrow$ Shake.Squeeze(prg, nb)
6 : **for** e from 0 to $(\tau - 1)$ **do**
7 : $\quad i^*[e] \leftarrow \mathsf{rnd}[e \cdot \log N, (e+1) \cdot \log N - 1]$ ▷ $\log N$ bits of rnd assigned to $i^*[e]$.
8 : $v_{\text{grinding}} \leftarrow \mathsf{rnd}[\tau \cdot \log N, \tau \cdot \log N + w - 1]$
9 : **return** $(\{i^*[e]\}_{e<\tau}, v_{\text{grinding}})$

ExpandChallengeEvaluationPoints(h) returns the challenge points on which are done the evaluations to be revealed to the verifier, see Step 3 of the Mirath protocol in Section 3.3. This routine is used in OpenRandomEvaluations and in ComputeEvaluations.

Algorithm 9 Challenge matrix $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{(mn-k)\times k}$.

ExpandChallengeMatrix(h)

1 : nb $\leftarrow (mn-k) \cdot k \cdot \lceil \mu \cdot \log q/8 \rceil$
2 : prg $\leftarrow$ Shake.Init()
3 : prg $\leftarrow$ Shake.Absorb(prg, h)
4 : $\boldsymbol{\Gamma} \leftarrow$ Shake.Squeeze(prg, nb)
5 : **return** $\boldsymbol{\Gamma}$

ExpandChallengeMatrix(h) returns the challenge matrix $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho\times(mn-k)}$, see Step 2 of the Mirath protocol in Section 3.3. It is used in Sign and in Verify.

4.6 Batched All-but-one Vector Commitments

This section details the algorithms of the batched all-but-one vector commitment (BAVC) scheme used in Mirath, see Section 3.1.

Algorithm 10 Routine BAVC.Commit.

BAVC.Commit(salt, rseed)

1 : $\mathsf{tree}[0] \leftarrow \mathsf{rseed}$
2 : **for** i from 0 to $(\tau \cdot N - 2)$ **do**
3 : $\quad(\mathsf{tree}[2i+1], \mathsf{tree}[2i+2]) \leftarrow \mathsf{ExpandSeed}_3(\mathsf{salt}, \mathsf{tree}[i], i)$
4 : **for** e from 0 to $(\tau - 1)$ **do**
5 : $\quad$**for** i from 0 to $(N - 1)$ **do**
6 : $\quad\quad\mathsf{seeds}[e][i] \leftarrow \mathsf{tree}[(\tau \cdot N - 1) + (i \cdot \tau + e)]$
7 : $\quad\quad\mathsf{commit}[e][i] \leftarrow \mathsf{Hash}_3(\mathsf{salt}, \mathsf{seeds}[e][i], (\tau \cdot N - 1) + i \cdot \tau + e)$
8 : $h_{\mathsf{com}} \leftarrow \mathsf{Hash}_3(\{\mathsf{commit}[e][i]\}_{e,i})$
9 : $\mathsf{key} \leftarrow \mathsf{tree} \parallel \mathsf{commit}$
10 : **return** $(\mathsf{seeds}, h_{\mathsf{com}}, \mathsf{key})$

Algorithm 11 Routine BAVC.Open.

BAVC.Open(key, $\{i^*[e]\}_e$)

1 : $(\mathsf{tree}, \mathsf{commit}) \leftarrow \mathsf{key}$
2 : $\mathsf{hidden} \leftarrow \{(\tau \cdot N - 1) + (i^*[e] \cdot \tau + e) : e \in \{0, \ldots, \tau - 1\}\}$
3 : $\mathsf{revealed} \leftarrow \{\tau \cdot N - 1, \ldots, 2 \cdot \tau \cdot N - 2\} \backslash \mathsf{hidden}$
4 : **for** i from $(\tau \cdot N - 2)$ downto 0 **do**
5 : $\quad$**if** $(2i+1) \in \mathsf{revealed}$ and $(2i+2) \in \mathsf{revealed}$ **then**
6 : $\quad\quad\mathsf{revealed} \leftarrow (\mathsf{revealed} \backslash \{2i+1, 2i+2\}) \cup \{i\}$
7 : **if** $|\mathsf{revealed}| > T_{\mathrm{open}}$
8 : $\quad$**return** $\perp$
9 : $\mathsf{path} \leftarrow \emptyset$
10 : **for** i from 0 to $(2 \cdot \tau \cdot N - 2)$ **do**
11 : $\quad$**if** $i \in \mathsf{revealed}$ **then**
12 : $\quad\quad\mathsf{path} \leftarrow \mathsf{path} \parallel \mathsf{tree}[i]$
13 : $\mathsf{commit}_{i^*} \leftarrow \mathsf{commit}[0][i^*[0]] \parallel \ldots \parallel \mathsf{commit}[\tau - 1][i^*[\tau - 1]]$
14 : $\pi_{\mathsf{BAVC}} \leftarrow \mathsf{path} \parallel \mathsf{commit}_{i^*}$
15 : **return** π_{BAVC}

Algorithm 10 describes BAVC.Commit, i.e. the Commit method of the BAVC scheme used in Mirath. This function outputs $(\mathsf{seeds}, h_{\mathsf{com}}, \mathsf{key})$, where $\mathsf{seeds} = \{\mathsf{seeds}[e][i]\}_{e<\tau, i<N}$ is a set of $\tau \cdot N$ seeds in $\{0,1\}^\lambda$, h_{com} is a commitment to seeds, and key contains a binary tree tree with leaves given by seeds and a set commit of $\tau \cdot N$ elements in $\{0,1\}^{2\lambda}$ containing one commit per seed in

seeds. BAVC.Commit is used during signing within the CommitWitnessPolynomials algorithm.

Algorithm 11 describes BAVC.Open, i.e. the Open method of the BAVC scheme used in Mirath, which outputs the opening π_{BAVC} for the leaves in tree with indexes in $\{i^*[e]\}_{e<\tau}$. BAVC.Open is used in OpenRandomEvaluations.

Algorithm 12 Routine BAVC.Reconstruct.

BAVC.Reconstruct$(\{i^*[e]\}_{e<\tau)\}}, \pi_{\mathsf{BAVC}}, \mathsf{salt})$

1 : $(\mathsf{path}, \mathsf{commit}_{i^*}) \leftarrow \pi_{\mathsf{BAVC}}$
2 : $\mathsf{hidden} \leftarrow \{(\tau \cdot N - 1) + (i^*[e] \cdot \tau + e) : e \in \{0, \ldots, \tau - 1\}\}$
3 : $\mathsf{revealed} \leftarrow \{\tau \cdot N - 1, \ldots, 2 \cdot \tau \cdot N - 2\} \backslash \mathsf{hidden}$
4 : **for** i from $(\tau \cdot N - 2)$ downto 0 **do**
5 : **if** $(2i + 1) \in \mathsf{revealed}$ and $(2i + 2) \in \mathsf{revealed}$ **then**
6 : $\mathsf{revealed} \leftarrow (\mathsf{revealed} \backslash \{2i + 1, 2i + 2\}) \cup \{i\}$
7 : $\mathsf{tree}[0], \ldots, \mathsf{tree}[2 \cdot \tau \cdot N - 2] \leftarrow \emptyset, \ldots, \emptyset$
8 : **for** i from 0 to $(2 \cdot \tau \cdot N - 2)$ **do**
9 : **if** $i \in \mathsf{revealed}$ **then**
10 : $(\mathsf{tree}[i], \mathsf{path}) \leftarrow \mathsf{path}$
11 : **if** $\mathsf{tree}[i] \neq \emptyset$ **and** $i < \tau \cdot N - 1$ **then**
12 : $(\mathsf{nodes}[2i + 1], \mathsf{nodes}[2i + 2]) \leftarrow \mathsf{ExpandSeed}_3(\mathsf{salt}, \mathsf{nodes}[i], i)$
13 : **for** e from 0 to $(\tau - 1)$ **do**
14 : **for** i from 0 to $(N - 1)$ **do**
15 : **if** $i \neq i^*[e]$ **then**
16 : $\mathsf{seeds}[e][i] \leftarrow \mathsf{nodes}[(\tau \cdot N - 1) + (i \cdot \tau + e)]$
17 : $\mathsf{com}[e][i] \leftarrow \mathsf{Hash}_3(\mathsf{salt}, \mathsf{seeds}[e][i], (\tau \cdot N - 1) + i \cdot \tau + e)$
18 : **else**
19 : $\mathsf{seeds}[e][i] \leftarrow \emptyset$
20 : $(\mathsf{com}[e][i] \parallel \mathsf{commit}_{i^*}) \leftarrow \mathsf{commit}_{i^*}$
21 : $h_{\mathsf{com}} \leftarrow \mathsf{Hash}_3(\{\mathsf{com}[e][i]\}_{e,i})$
22 : **return** $(h_{\mathsf{com}}, \mathsf{seeds})$

Algorithm 12 describes BAVC.Reconstruct, i.e. the Reconstruct method of the BAVC scheme used in Mirath. This function outputs $(h_{\mathsf{com}}, \mathsf{seeds})$, where h_{com} is the reconstruction of a commitment made for a set of N seeds, and seeds is the set of $N - \tau$ opened seeds $\{\mathsf{seeds}[e][i]\}_{0 \leq e < \tau, i \neq i^*[e]}$.

4.7 Commitment to Polynomials

This section details how to commit to the witness polynomials $P_S(X) = \boldsymbol{S} \cdot X + \boldsymbol{S}_{\mathsf{base}}$, $P_{C'}(X) = \boldsymbol{C'} \cdot X + \boldsymbol{C'}_{\mathsf{base}}$ and the masking polynomial $P_v(X) = \boldsymbol{v} \cdot X + \boldsymbol{v}_{\mathsf{base}}$

Algorithm 13 Commit to witness $(P_S,\ P_{C'})$ and masking P_v polynomials.

CommitWitnessPolynomials(salt, rseed, $\boldsymbol{S}, \boldsymbol{C}'$)

1 : (seeds, h_{com}, key) $\leftarrow$ BAVC.Commit(salt, rseed)
2 : **for** e from 0 to $(\tau - 1)$ **do**
3 : $(\boldsymbol{S}_{\mathsf{acc}}, \boldsymbol{C}'_{\mathsf{acc}}, \boldsymbol{v}) \leftarrow (\mathbf{0}, \mathbf{0}, \mathbf{0})$ $\triangleright$ $\boldsymbol{S}_{\mathsf{acc}} \in \mathbb{F}_q^{m \times r}$, $\boldsymbol{C}'_{\mathsf{acc}} \in \mathbb{F}_q^{r \times (n-r)}$, $\boldsymbol{v} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$
4 : $(\boldsymbol{S}_{\mathsf{base}}, \boldsymbol{C}'_{\mathsf{base}}, \boldsymbol{v}_{\mathsf{base}}) \leftarrow (\mathbf{0}, \mathbf{0}, \mathbf{0})$ $\triangleright$ $\boldsymbol{S}_{\mathsf{base}} \in \mathbb{F}_{q^\mu}^{m \times r}$, $\boldsymbol{C}'_{\mathsf{base}} \in \mathbb{F}_{q^\mu}^{r \times (n-r)}$, $\boldsymbol{v}_{\mathsf{base}} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$
5 : **for** i from 0 to $(N - 1)$ **do**
6 : $\left(\boldsymbol{S}_{\mathsf{rnd}}, \boldsymbol{C}'_{\mathsf{rnd}}, \boldsymbol{v}_{\mathsf{rnd}}\right) \leftarrow$ ExpandSeedShares(salt, seeds[e][i])
7 : $\left(\boldsymbol{S}_{\mathsf{acc}}, \boldsymbol{C}'_{\mathsf{acc}}, \boldsymbol{v}_{\mathsf{acc}}\right)$ += $\left(\boldsymbol{S}_{\mathsf{rnd}}, \boldsymbol{C}'_{\mathsf{rnd}}, \boldsymbol{v}_{\mathsf{rnd}}\right)$
8 : $\left(\boldsymbol{S}_{\mathsf{base}}, \boldsymbol{C}'_{\mathsf{base}}, \boldsymbol{v}_{\mathsf{base}}\right)$ -= $\left(\phi(i) \cdot \boldsymbol{S}_{\mathsf{rnd}}, \phi(i) \cdot \boldsymbol{C}'_{\mathsf{rnd}}, \phi(i) \cdot \boldsymbol{v}_{\mathsf{rnd}}\right)$
$\triangleright$ $\phi : \{0, \ldots, q^\mu - 1\} \to \mathbb{F}_{q^\mu}$
9 : aux[e] $\leftarrow \left(\boldsymbol{S} - \boldsymbol{S}_{\mathsf{acc}}, \boldsymbol{C}' - \boldsymbol{C}'_{\mathsf{acc}}\right)$
10 : base[e] $\leftarrow \left(\boldsymbol{S}_{\mathsf{base}}, \boldsymbol{C}'_{\mathsf{base}}, \boldsymbol{v}_{\mathsf{base}}\right)$
11 : $\boldsymbol{v}$[e] $\leftarrow \boldsymbol{v}_{\mathsf{acc}}$
12 : $h_{\mathrm{sh}} \leftarrow \mathsf{Hash}_1(\mathsf{salt}, h_{\mathsf{com}}, \mathsf{aux}[0], \ldots, \mathsf{aux}[\tau - 1])$
13 : **return** (base, $\boldsymbol{v}$, h_{sh}, key, aux)

as indicated at Step 1 of Mirath protocol. It also describes routines to open (and verify) some random evaluations of the committed polynomials, see Step 3 of the Mirath protocol Section 3.3.

Algorithm 14 Open random evaluations.

OpenRandomEvaluations(key, h_{piop})

1 : ctr $\leftarrow 0$
2 : `retry:`
3 : $(\{i^*[e]\}_{e<\tau}, v_{\mathrm{grinding}}) \leftarrow$ ExpandChallengeEvaluationPoints(h_{piop}, ctr)
4 : $\triangleright$ $i^*[e] \in \{0, \ldots, (N-1)\}$
5 : $\pi_{\mathsf{BAVC}} \leftarrow$ BAVC.Open(key, $\{i^*[e]\}_{e<\tau}$)
6 : **if** $\pi_{\mathsf{BAVC}} = \perp$ or $v_{\mathrm{grinding}} \neq 0$ **then**
7 : ctr $\leftarrow$ ctr $+ 1$
8 : **goto** `retry`
9 : **return** (ctr, π_{BAVC})

Algorithm 13 describes the routine CommitWitnessPolynomials that commits to τ different triples of pseudorandom polynomials $(P_S(X), P_{C'}(X), P_v(X)$. This

routine follows the description given in the subsection Committing to Witness Polynomials, and is used Sign.

Algorithm 14 details the routine OpenRandomEvaluations used in Sign to output $(\mathsf{ctr}, \pi_{\mathsf{BAVC}})$, which contains the necessary information to compute the evaluations $\boldsymbol{S}_{\mathsf{eval}} = P_S(\mathsf{point}[e])$, $\boldsymbol{C}'_{\mathsf{eval}} = P_C(\mathsf{point}[e])$, and $\boldsymbol{v}_{\mathsf{eval}} = P_v(\mathsf{point}[e])$ for any $e < \tau$ and to check the consistency of the commitments made at Step 1 of the Mirath protocol. The value π_{BAVC} is the output of the subroutine BAVC.Open, while ctr incorporates the output from the grinding and proof-of-work optimizations.

Algorithm 15 details the routine ComputeEvaluations used in Verify to compute the evaluations $\boldsymbol{S}_{\mathsf{eval}} = P_S(\mathsf{point}[e])$, $\boldsymbol{C}'_{\mathsf{eval}} = P_C(\mathsf{point}[e])$ and $\boldsymbol{v}_{\mathsf{eval}} = P_v(\mathsf{point}[e])$ for any $e < \tau$, and to check the consistency of the commitment made at Step 1 of the Mirath protocol.

Algorithm 15 Compute evaluations of P_S, $P_{C'}$ and P_v on the opened points.

ComputeEvaluations$(\mathsf{salt}, \mathsf{ctr}, h_{\mathrm{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux})$

$(\{i^*[e]\}_{e<\tau}, v_{\mathrm{grinding}}) \leftarrow$ ExpandChallengeEvaluationPoints$(h_{\mathrm{piop}}, \mathsf{ctr})$
$h_{\mathsf{com}}, \mathsf{seeds} \leftarrow$ BAVC.Reconstruct$(\{i^*[e]\}_{e<\tau)\}}, \pi_{\mathsf{BAVC}}, \mathsf{salt})$
$h_{\mathrm{sh}} \leftarrow \mathsf{Hash}_1(\mathsf{salt}, h_{\mathsf{com}}, \mathsf{aux}[0], \ldots, \mathsf{aux}[\tau-1])$
for e from 0 to $(\tau-1)$ **do**
  $(\boldsymbol{S}_{\mathsf{eval}}[e], \boldsymbol{C}'_{\mathsf{eval}}[e], \boldsymbol{v}_{\mathsf{eval}}[e]) = (\mathbf{0}, \mathbf{0}, \mathbf{0})$
    $\triangleright$ $(\boldsymbol{S}_{\mathsf{eval}}[e], \boldsymbol{C}'_{\mathsf{eval}}[e], \boldsymbol{v}_{\mathsf{eval}}[e]) \in \mathbb{F}_{q^\mu}^{m\times r} \times \mathbb{F}_{q^\mu}^{r\times(n-r)} \times \mathbb{F}_{q^\mu}^{\rho\times 1}$
  for i from 0 to $N-1$ **do**
    if $i \neq i^*[e]$ **then**
      $(\boldsymbol{S}_{\mathsf{rnd}}, \boldsymbol{C}_{\mathsf{rnd}}, \boldsymbol{v}_{\mathsf{rnd}}) \leftarrow$ ExpandSeedShares$(\mathsf{salt}, \mathsf{seeds}[e][i])$
      $a \leftarrow \phi(i^*[e]) - \phi(i)$ $\triangleright$ $\phi : \{0, \ldots, q^\mu - 1\} \to \mathbb{F}_{q^\mu}$
      $(\boldsymbol{S}_{\mathsf{eval}}[e], \boldsymbol{C}'_{\mathsf{eval}}[e], \boldsymbol{v}_{\mathsf{eval}}[e])$ += $\left(a\boldsymbol{S}_{\mathsf{rnd}}, a\boldsymbol{C}'_{\mathsf{rnd}}, a\boldsymbol{v}_{\mathsf{rnd}}\right)$
  $\mathsf{point}[e] \leftarrow \phi(i^*[e])$
  $(\boldsymbol{S}_{\mathsf{aux}}, \boldsymbol{C}'_{\mathsf{aux}}) \leftarrow \mathsf{aux}[e]$
  $(\boldsymbol{S}_{\mathsf{eval}}[e], \boldsymbol{C}'_{\mathsf{eval}}[e])$ += $(\mathsf{point}[e] \cdot \boldsymbol{S}_{\mathsf{aux}}, \mathsf{point}[e] \cdot \boldsymbol{C}'_{\mathsf{aux}})$
  $\mathsf{eval}[e] \leftarrow (\boldsymbol{S}_{\mathsf{eval}}[e], \boldsymbol{C}'_{\mathsf{eval}}[e], \boldsymbol{v}_{\mathsf{eval}}[e])$
return $\left(\mathsf{eval}, \mathsf{point}, h_{\mathrm{sh}}, v_{\mathrm{grinding}}\right)$

4.8 Computation of Polynomial Proof

This section describes how to compute and recompute the polynomial proof $P_\alpha(X)$, see Step 2 of the Mirath protocol in Section 3.3.

Algorithm 16 describes the routine ComputePolynomialProof used in Sign to compute the τ polynomial proof $P_\alpha(X) = \boldsymbol{\alpha}_{\mathsf{mid}} \cdot X + \boldsymbol{\alpha}_{\mathsf{base}}$ from the τ committed

polynomials $P_S(X) = \boldsymbol{S} \cdot X + \boldsymbol{S}_{\mathsf{base}}, P_C(X) = \boldsymbol{C}' \cdot X + \boldsymbol{C}'_{\mathsf{base}}, P_v(X) = \boldsymbol{v} \cdot X + \boldsymbol{v}_{\mathsf{base}}$ and the matrices $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (m \cdot n - k)}$ and $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k) \times k}$.

Algorithm 16 Computation of the polynomial P_α.

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

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

Algorithm 17 describes RecomputePolynomialProof, used in Verify to recover the polynomial proofs $P_\alpha(X)$ from the different evaluations $\boldsymbol{S}_{\mathsf{eval}} = P_S(\mathsf{point}[e])$, $\boldsymbol{C}'_{\mathsf{eval}} = P_C(\mathsf{point}[e])$ and $\boldsymbol{v}_{\mathsf{eval}} = P_v(\mathsf{point}[e])$ with a given $\mathsf{point}[0], \ldots, \mathsf{point}[\tau - 1] \in S \subset \mathbb{F}_{q^\mu}$.

Algorithm 17 Reconstruction of the polynomial P_α.

$\mathsf{RecomputePolynomialProof}(\mathsf{point}, \mathsf{eval}, \boldsymbol{\Gamma}, (\boldsymbol{H}', \boldsymbol{y}), \boldsymbol{\alpha}_{\mathsf{mid}})$

1: $\boldsymbol{S}_{\mathsf{eval}}, \boldsymbol{C}'_{\mathsf{eval}}, \boldsymbol{v}_{\mathsf{eval}} \leftarrow \mathsf{eval}$
2: $\boldsymbol{E}_{\mathsf{eval}} \leftarrow (\mathsf{point} \cdot \boldsymbol{S}_{\mathsf{eval}} \mid \boldsymbol{S}_{\mathsf{eval}} \boldsymbol{C}'_{\mathsf{eval}})$ ▷ $\boldsymbol{E}_{\mathsf{eval}} \in \mathbb{F}_{q^\mu}^{m \times n}$
3: $\boldsymbol{e} \leftarrow \mathrm{vec}(\boldsymbol{E}_{\mathsf{eval}})$ ▷ $\boldsymbol{e} \in \mathbb{F}_{q^\mu}^{mn}$ with the entries of $\boldsymbol{E}_{\mathsf{eval}}$ in a column-major order
4: $(\boldsymbol{e}_A, \boldsymbol{e}_B) \leftarrow \mathsf{Split}(\boldsymbol{e})$ ▷ $\boldsymbol{e}_A \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}, \boldsymbol{e}_B \times \mathbb{F}_{q^\mu}^{k \times 1}$
5: $\boldsymbol{\alpha}_{\mathsf{eval}} \leftarrow \boldsymbol{\Gamma}(\boldsymbol{e}_A + \boldsymbol{H}' \boldsymbol{e}_B - \boldsymbol{y} \cdot \mathsf{point}^2) + \boldsymbol{v}_{\mathsf{eval}}$ ▷ $\boldsymbol{\alpha}_{\mathsf{eval}} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$
6: $\boldsymbol{\alpha}_{\mathsf{base}} \leftarrow \boldsymbol{\alpha}_{\mathsf{eval}} - \boldsymbol{\alpha}_{\mathsf{mid}} \cdot \mathsf{point}$
7: **return** $\boldsymbol{\alpha}_{\mathsf{base}}$

4.9 Parsing of the Signature

In Algorithm 18 and Algorithm 19 we give the details of packing/unpacking the signature in Mirath. This packing is performed tightly in order to efficiently avoid

Algorithm 18 Unparse signature.

UnparseSignature $(\mathsf{salt}, \mathsf{ctr}, h_{\text{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux}, \boldsymbol{\alpha}_{\mathsf{mid}})$

$\mathtt{nbits} \leftarrow 2\lambda + 64 + 2\lambda + T_{\text{open}}\lambda + 2\tau\lambda + \tau(mr + r(n-r) + \rho\mu)\log q$
$\mathsf{nb} \leftarrow \lceil \mathtt{nbits}/8 \rceil$
$\sigma \leftarrow \mathbf{0}_{\mathsf{nb}}$ ▷ An array of nb zero bytes
$\sigma[0, \lambda(4 + T_{\text{open}} + 2\tau) + 63] \leftarrow (\mathsf{salt} \parallel \mathtt{bin}_{64}(\mathsf{ctr}) \parallel h_{\text{piop}} \parallel \pi_{\mathsf{BAVC}})$
▷ $\mathtt{bin}_{64}(\mathsf{ctr})$ is the 64-bit representation of ctr.
$\mathtt{i}_b \leftarrow \lambda(4 + T_{\text{open}} + 2\tau) + 64$
for e from 0 to $(\tau - 1)$ **do**
  $(\boldsymbol{S}_{\mathsf{aux}} \parallel \boldsymbol{C}'_{\mathsf{aux}}) \leftarrow \mathsf{aux}[e]$ ▷ $\boldsymbol{S}_{\mathsf{aux}} \in \mathbb{F}_q^{m\times r}$, $\boldsymbol{C}'_{\mathsf{aux}} \in \mathbb{F}_q^{r\times(n-r)}$
  $\sigma, \mathtt{i}_b \leftarrow \mathsf{UnparseMatrix}(\sigma, \boldsymbol{S}_{\mathsf{aux}}, \mathtt{i}_b, \mathbb{F}_q^{m\times r})$
  $\sigma, \mathtt{i}_b \leftarrow \mathsf{UnparseMatrix}(\sigma, \boldsymbol{C}'_{\mathsf{aux}}, \mathtt{i}_b, \mathbb{F}_q^{r\times(n-r)})$
  $\sigma, \mathtt{i}_b \leftarrow \mathsf{UnparseMatrix}(\sigma, \boldsymbol{\alpha}_{\mathsf{mid}}[e], \mathtt{i}_b, \mathbb{F}_{q^\mu}^{\rho\times 1})$
return σ

trivial forgeries, and whenever the bitsize of the signature is not a multiple of eight, we set the free bits in the last byte of the signature to zeros and check them during the verification.

Algorithm 19 Parse signature.

ParseSignature(σ)

$\mathsf{salt} \leftarrow \sigma[0, 2\lambda - 1]$
$\mathsf{ctr} \leftarrow \sigma[\lambda, \lambda + 63]$
$h_{\text{piop}} \leftarrow \sigma[\lambda + 64, 3\lambda + 63]$
$\pi_{\mathsf{BAVC}} \leftarrow \sigma[3\lambda + 64, \lambda(3 + T_{open} + 2\tau) + 63]$
$\mathtt{i}_b \leftarrow \lambda(3 + T_{open} + 2\tau) + 64$
for e from 0 to $(\tau - 1)$ **do**
  $\boldsymbol{S}_{\mathsf{aux}}, \mathtt{i}_b \leftarrow \mathsf{ParseMatrix}(\sigma, \mathtt{i}_b, \mathbb{F}_q^{m\times r})$ ▷ $\boldsymbol{S}_{\mathsf{aux}} \in \mathbb{F}_q^{m\times r}$
  $\boldsymbol{C}'_{\mathsf{aux}}, \mathtt{i}_b \leftarrow \mathsf{ParseMatrix}(\sigma, \mathtt{i}_b, \mathbb{F}_q^{r\times(n-r)})$ ▷ $\boldsymbol{C}'_{\mathsf{aux}} \in \mathbb{F}_q^{r\times(n-r)}$
  $\mathsf{aux}[e] \leftarrow (\boldsymbol{S}_{\mathsf{aux}} \parallel \boldsymbol{C}'_{\mathsf{aux}})$
  $\boldsymbol{\alpha}_{\mathsf{mid}}, \mathtt{i}_b \leftarrow \mathsf{ParseMatrix}(\sigma, \mathtt{i}_b, \mathbb{F}_{q^\mu}^{\rho\times 1})$
return $(\mathsf{salt}, \mathsf{ctr}, h_{\text{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux}, \boldsymbol{\alpha}_{\mathsf{mid}})$

4.10 Key Generation

The key generation algorithm is described in Algorithm 21. The bit size of the public key is $|\mathsf{seed}_{\mathsf{pk}}| + |\boldsymbol{y}| = \lambda + (mn - k)\log q$ and the bit size of the secret key is $|\mathsf{seed}_{\mathsf{pk}}| + |\mathsf{seed}_{\mathsf{sk}}| = 2\lambda$.

Algorithm 20 Routine ComputeY.

ComputeY$(\boldsymbol{S}, \boldsymbol{C}', \boldsymbol{H}')$

1 : $\boldsymbol{E} \leftarrow (\boldsymbol{S} \mid \boldsymbol{S}\boldsymbol{C}')$ ▷ $\boldsymbol{E} \in \mathbb{F}_q^{m \times n}$
2 : $\boldsymbol{e} \leftarrow \mathrm{vec}(\boldsymbol{E})$ ▷ $\boldsymbol{e} \in \mathbb{F}_q^{mn}$ with the entries of $\boldsymbol{E}$ in a column-major order
3 : $(\boldsymbol{e}_A, \boldsymbol{e}_B) \leftarrow \mathsf{Split}(\boldsymbol{e})$ ▷ $\boldsymbol{e}_A \in \mathbb{F}_q^{(mn-k)\times 1}, \boldsymbol{e}_B \in \mathbb{F}_q^{k \times 1}$
4 : $\boldsymbol{y} \leftarrow \boldsymbol{e}_A + \boldsymbol{H}'\boldsymbol{e}_B$ ▷ $\boldsymbol{y} \in \mathbb{F}_q^{(mn-k)\times 1}$
5 : **return** $\boldsymbol{y}$

Algorithm 21 Key generation algorithm.

KeyGen()

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

The key decompressing routines are shown in Algorithms 22 and 23.

Algorithm 22 Decompress public key routine.

DecompressPK(pk)

1 : $\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y} \leftarrow \mathsf{pk}$
2 : $\boldsymbol{H}' \leftarrow \mathsf{ExpandSeedPublicMatrix}(\mathsf{seed}_{\mathsf{pk}})$
3 : **return** $(\boldsymbol{H}', \boldsymbol{y})$

Algorithm 23 Decompress secret key routine.

DecompressSK(sk)

1 : $\mathsf{seed}_{\mathsf{sk}}, \mathsf{seed}_{\mathsf{pk}} \leftarrow \mathsf{sk}$
2 : $\boldsymbol{H}' \leftarrow \mathsf{ExpandSeedPublicMatrix}(\mathsf{seed}_{\mathsf{pk}})$
3 : $(\boldsymbol{S}, \boldsymbol{C}') \leftarrow \mathsf{ExpandSeedSecretMatrices}(\mathsf{seed}_{\mathsf{sk}})$
4 : $\boldsymbol{y} \leftarrow \mathsf{ComputeY}(\boldsymbol{S}, \boldsymbol{C}', \boldsymbol{H}')$
5 : $\mathsf{pk} \leftarrow (\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y})$
6 : **return** $(\boldsymbol{H}', \mathsf{pk}, \boldsymbol{S}, \boldsymbol{C}')$

4.11 Signing

Algorithm 24 specifies Mirath's signing algorithm Sign. The maximum bit size of a Mirath signature σ is

$$|\sigma| = \underbrace{2\lambda}_{\mathsf{salt}} + \underbrace{64}_{\mathsf{ctr}} + \underbrace{2\lambda}_{h_{\mathrm{piop}}} + \underbrace{\lambda \cdot T_{\mathrm{open}}}_{\mathsf{path}\ (\text{in}\ \pi_{\mathsf{BAVC}})} + \underbrace{\tau \cdot 2\lambda}_{\mathsf{commit}_{i^*}\ (\text{in}\ \pi_{\mathsf{BAVC}})} + \\ + \tau \cdot \Big(\underbrace{[rm + r(n-r)] \cdot \log_2 q}_{\mathsf{aux}} + \underbrace{\rho\mu \cdot \log_2 q}_{\boldsymbol{\alpha}_{\mathsf{mid}}} \Big).$$

Algorithm 24 Signing algorithm.

$\mathsf{Sign}(\mathsf{sk}, \mathsf{msg})$

// Step 0: Initialization.

$(\boldsymbol{H}', \mathsf{pk}, \boldsymbol{S}, \boldsymbol{C}') \leftarrow \mathsf{DecompressSK}(\mathsf{sk})$

$\mathsf{salt} \leftarrow \mathsf{TRG.Seed}(2\lambda)$

$\mathsf{rseed} \leftarrow \mathsf{TRG.Seed}(\lambda)$

// Step 1: Build & Commit to Witness Polynomials.

$(\mathsf{base}, \boldsymbol{v}, h_{\mathrm{sh}}, \mathsf{key}, \mathsf{aux}) \leftarrow \mathsf{CommitWitnessPolynomials}(\mathsf{salt}, \mathsf{rseed}, \boldsymbol{S}, \boldsymbol{C}')$

// Step 2: Compute the polynomial proof $P_\alpha(X)$.

$\boldsymbol{\Gamma} \leftarrow \mathsf{ExpandChallengeMatrix}(h_{\mathrm{sh}})$ ▷ $\boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

for e from 0 to $(\tau - 1)$ **do**

  $(\boldsymbol{\alpha}_{\mathsf{mid}}[e], \boldsymbol{\alpha}_{\mathsf{base}}[e]) \leftarrow \mathsf{ComputePolynomialProof}\left(\mathsf{base}[e], \boldsymbol{v}[e], \boldsymbol{S}, \boldsymbol{C}', \boldsymbol{\Gamma}, \boldsymbol{H}'\right)$

$h_{\mathrm{piop}} \leftarrow \mathsf{Hash}_2\left(\mathsf{pk}, \mathsf{salt}, \mathsf{msg}, h_{\mathrm{sh}}, \boldsymbol{\alpha}_{\mathsf{mid}}[0], \boldsymbol{\alpha}_{\mathsf{base}}[0], \ldots, \boldsymbol{\alpha}_{\mathsf{mid}}[\tau-1], \boldsymbol{\alpha}_{\mathsf{base}}[\tau-1]\right)$

// Step 3: Open random evaluations of the polynomials $P_S(X)$, $P_C(X)$ and $P_v(X)$.

$(\mathsf{ctr}, \pi_{\mathsf{BAVC}}) \leftarrow \mathsf{OpenRandomEvaluations}(\mathsf{key}, h_{\mathrm{piop}})$

$\sigma \leftarrow \mathsf{UnparseSignature}\left(\mathsf{salt}, \mathsf{ctr}, h_{\mathrm{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux}, \boldsymbol{\alpha}_{\mathsf{mid}}\right)$

return σ

4.12 Verification

Algorithm 25 specifies Mirath's verification algorithm Verify.

Algorithm 25 Verification algorithm.

Verify$(\mathsf{pk}, \sigma, \mathsf{msg})$

// Step 0: Initialization (parsing and expansion).

$(\mathsf{salt}, \mathsf{ctr}, h_{\mathrm{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux}, \boldsymbol{\alpha}_{\mathsf{mid}}) \leftarrow \mathsf{ParseSignature}(\sigma)$

$(\boldsymbol{H}', \boldsymbol{y}) \leftarrow \mathsf{DecompressPK}(\mathsf{pk})$

// Step 1: Computing Opened Evaluations.

$(\mathsf{evals}, \mathsf{point}, h_{\mathrm{sh}}, v_{\mathrm{grinding}}) \leftarrow \mathsf{ComputeEvaluations}(\mathsf{salt}, \mathsf{ctr}, h_{\mathrm{piop}}, \pi_{\mathsf{BAVC}}, \mathsf{aux})$

// Step 2: Recomputation of the Polynomial Proof $P_\alpha(X)$.

$\boldsymbol{\Gamma} \leftarrow \mathsf{ExpandChallengeMatrix}(h_{\mathrm{piop}})$ $\quad \triangleright\ \boldsymbol{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

for e from 0 to $(\tau - 1)$ **do**

  $\boldsymbol{\alpha}_{\mathsf{base}}[e] \leftarrow \mathsf{RecomputePolynomialProof}(\mathsf{point}[e], \mathsf{evals}[e], \boldsymbol{\Gamma}, (\boldsymbol{H}', \boldsymbol{y}), \boldsymbol{\alpha}_{\mathsf{mid}}[e])$

// Step 3: Verification.

$h'_{\mathrm{piop}} \leftarrow \mathsf{Hash}_2(\mathsf{pk}, \mathsf{salt}, \mathsf{msg}, h_{\mathrm{sh}}, \boldsymbol{\alpha}_{\mathsf{mid}}[0], \boldsymbol{\alpha}_{\mathsf{base}}[0], \ldots, \boldsymbol{\alpha}_{\mathsf{mid}}[\tau-1], \boldsymbol{\alpha}_{\mathsf{base}}[\tau-1])$

return $(h_{\mathrm{piop}} \stackrel{?}{=} h'_{\mathrm{piop}})$ and $(v_{\mathrm{grinding}} \stackrel{?}{=} 0)$

5 Parameter Sets

We provide several parameter sets using the nomenclature Mirath-Xy-z where $\mathsf{X} \in \{1, 3, 5\}$ denotes the security level, $\mathsf{y} \in \{\mathsf{a}, \mathsf{b}\}$ corresponds to parameters using respectively $q = 16$ and $q = 2$ for the base field $\mathbb{F}_q$ and $\mathsf{z} \in \{\mathsf{short}, \mathsf{fast}\}$ refers to size/performance trade-off considered for the parameter set.

5.1 MinRank Parameters

MinRank parameters used in Mirath are given in Tables 4 and 5. The security of Mirath against classical MinRank attacks is estimated using the MinRank estimator from the CryptographicEstimators V2.0.0 [2] [18], which considers all the classical attacks described in Section 8. The bit security estimates given in Tables 5 and 4 are computed by configuring the estimator with all values default but the matrix multiplication constant (ω in Section 8.2) begin set to 2.81.

Instance	q	m	n	k	r	classical bit security
Mirath-1a	16	16	16	143	4	158
Mirath-3a	16	19	19	195	5	225
Mirath-5a	16	22	22	255	6	301

Table 4. MinRank parameters used in Mirath-a ($q = 16$)

Instance	q	m	n	k	r	classical bit security
Mirath-1b	2	42	42	1443	4	158
Mirath-3b	2	50	50	2024	5	224
Mirath-5b	2	56	56	2499	6	289

Table 5. MinRank parameters used in Mirath-b ($q = 2$)

[2] Code available at https://github.com/Crypto-TII/CryptographicEstimators.

5.2 Protocol Parameters

The protocol related parameters used in Mirath are given in Table 6.

Instance	q	μ	ρ	τ	N	T_{open}	w
Mirath-1a-short	16	3	11	11	2^{12}	116	7
Mirath-1b-short	2	12					
Mirath-1a-fast	16	2	16	17	2^{8}	118	9
Mirath-1b-fast	2	8					
Mirath-3a-short	16	3	16	17	2^{12}	174	5
Mirath-3b-short	2	12					
Mirath-3a-fast	16	2	24	26	2^{8}	184	10
Mirath-3b-fast	2	8					
Mirath-5a-short	16	3	22	23	2^{12}	232	3
Mirath-5b-short	2	12					
Mirath-5a-fast	16	2	32	36	2^{8}	244	4
Mirath-5b-fast	2	8					

Table 6. MPC parameters used in Mirath ($q = 16$ and $q = 2$)

5.3 Key and Signature Sizes

Public key, secret key and signatures size of Mirath are given in Tables 7 and 8.

Instance	sk size [B]	pk size [B]	σ size [B]
Mirath-1a-short	32	73	3,078
Mirath-1a-fast	32	73	3,728
Mirath-3a-short	48	107	6,907
Mirath-3a-fast	48	107	8,537
Mirath-5a-short	64	147	12,413
Mirath-5a-fast	64	147	15,504

Table 7. Keys and signature sizes of Mirath-a ($q = 16$)

Instance	sk size [B]	pk size [B]	σ size [B]
Mirath-1b-short	32	57	2,902
Mirath-1b-fast	32	57	3,456
Mirath-3b-short	48	84	6,514
Mirath-3b-fast	48	84	7,936
Mirath-5b-short	64	112	11,620
Mirath-5b-fast	64	112	14,262

Table 8. Keys and signature sizes of Mirath-b ($q = 2$)

6 Implementation and Performance Analysis

This section provides performance measures of our implementations of Mirath.

Benchmark platform. The benchmarks were performed on a machine running Ubuntu Server 22.04.5 LTS, equipped with an Intel 13th-generation Intel (R) Core(TM) i9-13900K CPU running at 3000MHz and 64GB of RAM. All the experiments were performed with Hyper-Threading, Turbo Boost, and SpeedStep features disabled. The scheme has been compiled with GCC compiler (version 11.4.0) and uses the XKCP library.

The results were obtained by computing the mean from i random instances. To minimize biases from background tasks running on the benchmark platform, each instance has been repeated i times and averaged. Results were computed using $i = 25$ for the optimized implementation and $i = 5$ for the reference implementation. Additionally, all the parameter sets run without increasing the stack memory, except Mirath-5a-short and Mirath-5b-short, which requires increasing the stack memory to 9372 kilobytes (e.g. by running `ulimit -s 9372`).

Remark on the instantiation of Mirath. The performance profile of Mirath (and more generally any MPCitH based schemes) is highly dependent on the performances of the underlying symmetric primitives. In Mirath, PRG are instantiated using either AES or SHAKE while hash functions are instantiated using SHA3. One should note that different choices would lead to significant differences in performances. We may provide additional benchmarks with different symmetric primitive choices in the future.

6.1 Reference Implementation

The performances of our reference implementations on the aforementioned benchmark platform are reported in Tables 9 and 10 respectively. One should note that our reference implementation is only provided to help understanding the scheme and as such is not representative of the performance that the scheme can achieve, we defer the reader to Tables 11 and 12 for the performances of our optimized implementation. In addition, our reference implementation is not implemented in a constant-time way.

Instance	Keygen	Sign	Verify
Mirath-1a-short	0.3 M	16.3 B	26.8 B
Mirath-1a-fast	0.3 M	1.6 B	2.6 B
Mirath-3a-short	0.6 M	79.0 B	134 B
Mirath-3a-fast	0.6 M	6.8 B	11.0 B
Mirath-5a-short	1.1 M	95.2 B	157 B
Mirath-5a-fast	1.1 M	9.5 B	15.3 B

Table 9. Performances of Mirath-a (Reference) in Millions (M) and Billions (B) of CPU Cycles.

Instance	Keygen	Sign	Verify
Mirath-1b-short	1.8 M	18.3 B	30.9 B
Mirath-1b-fast	1.7 M	1.6 B	2.6 B
Mirath-3b-short	3.7 M	69.5 B	115 B
Mirath-3b-fast	3.7 M	6.8 B	11.0 B
Mirath-5b-short	6.1 M	94.5 B	156 B
Mirath-5b-fast	6.1 M	8.4 B	13.0 B

Table 10. Performances of Mirath-b (Reference) in Thousands (K) and Billions (B) of CPU Cycles.

6.2 Optimized Implementation

The performance of our the optimized implementations on the aforementioned benchmark platform are described in Tables 11 and 12 respectively. Our optimized implementation has been implemented in a constant time way whenever relevant, and as such, the running time is expected to not leak any information concerning sensitive data.

Instance	Keygen	Sign	Verify
Mirath-1a-short	0.2 M	97.3 M	94.6 M
Mirath-1a-fast	0.2 M	11.2 M	9.8 M
Mirath-3a-short	0.2 M	361 M	412 M
Mirath-3a-fast	0.2 M	31.8 M	34.5 M
Mirath-5a-short	0.4 M	668 M	614 M
Mirath-5a-fast	0.4 M	65.6 M	65.1 M

Table 11. Performances of Mirath-a (Optimized) in Millions of CPU Cycles.

Instance	Keygen	Sign	Verify
Mirath-1b-short	0.6 M	107 M	102 M
Mirath-1b-fast	0.6 M	14.8 M	12.2 M
Mirath-3b-short	1.2 M	300 M	332 M
Mirath-3b-fast	1.2 M	53.2 M	52.5 M
Mirath-5b-short	1.9 M	698 M	629 M
Mirath-5b-fast	1.9 M	96.0 M	91.6 M

Table 12. Performances of Mirath-b (Optimized) in Millions of CPU Cycles.

6.3 Known Answer Test Values

Known Answer Test (KAT) values have been generated using the script provided by the NIST and can be retrieved in the KATs/ folder. Both reference and optimized implementations generate the same KATs.

7 Security Analysis

7.1 Security Proof

In this section, we provide a security proof for the Mirath scheme. One should note that this proof relies on generic PRGs, Hash functions and commitment schemes and as such does not encompass the specific choices made in order to instantiate these primitives while implementing the scheme.

Theorem 1. *Let the PRG used be (t, ϵ_{PRG})-secure, and ϵ_{SMR} the advantage an adversary has over the Syndrome MinRank problem. Consider $\mathsf{Hash}_1, \mathsf{Hash}_2, \mathsf{Hash}_3$ behave as random oracles, with an output of 2λ bits. If an adversary makes q_i queries to Hash_i and q_S queries to the signing oracle, then the probability for him to produce a forgery for the Mirath Signature Scheme is bounded by:*

$$\begin{aligned}\mathsf{Pr}[\mathsf{Forge}] \leq & \frac{3 \cdot (q' + (\tau \cdot N + 1) \cdot q_S)^2}{2 \cdot 2^{2\lambda}} + \frac{q_S \cdot (q_S + 3q')}{2^{2\lambda}} \\ & + q_S \cdot \tau \cdot \epsilon_{PRG} + q' \cdot 2^{-w} \cdot \left(\frac{2}{N}\right)^{\tau} + q' \cdot \tau \cdot \frac{1}{q^{\mu \cdot \rho}} + \epsilon_{SMR}\end{aligned}$$

where $q' = \max(q_1, q_2, q_3)$ and τ is the number of repetitions of the signature.

Proof. In this proof, we will adopt a game hopping strategy in order to find the upper bound. The first game will be the access to the standard signing oracle by the adversary $\mathcal{A}$. We will then game hop in order to eliminate the cases where collisions happen, and, through some other games, we will manage to find an upper bound. We note $\mathsf{Pr}_i[\mathsf{Forge}]$ the probability of forgery when considering game i. The aim of the proof is to find an upper bound on $\mathsf{Pr}_1[\mathsf{Forge}]$.

- **Game 1.** This is the interaction between $\mathcal{A}$ and the real signature scheme. KeyGen generates $(\boldsymbol{H}, \boldsymbol{y}, \boldsymbol{S}, \boldsymbol{C'})$ and $\mathcal{A}$ receives $(\boldsymbol{H}, \boldsymbol{y})$. $\mathcal{A}$ can make queries to each Hash_i independently, and can make signing queries. At the end of the attack, $\mathcal{A}$ outputs a message/signature pair, (m, σ). The event Forge happens when the message output by $\mathcal{A}$ was not previously used in a query to the signing oracle.
- **Game 2.** In this game, we add a condition to the success of the attacker. The condition we add is that if there is a collision between outputs of Hash_1, or Hash_2, or Hash_3, then, the forgery is not valid. The first step is to look at the number of times every Hash_i is called when calling the signing oracle. The signing oracle contains one call to Hash_1 and Hash_2, and $(\tau \cdot N + 1)$ to Hash_3. The number of queries to Hash_1, to Hash_2, or to Hash_3 is then bounded from above by $q' + (\tau \cdot N + 1) \cdot q_S$, where q_i is the number of queries made by $\mathcal{A}$ to Hash_i, $q' = \max\{q_1, q_2, q_3\}$, q_S is the number of queries to the signing oracle. We thus have:

$$|\mathsf{Pr}_1[\mathsf{Forge}] - \mathsf{Pr}_2[\mathsf{Forge}]| \leq \frac{3 \cdot (q' + (\tau \cdot N + 1) \cdot q_S)^2}{2 \cdot 2^{2\lambda}}$$

– **Game 3.** The attacker now fails if the inputs to any of the Hash_i has already appeared in a previous query. If that happens, this means that at least the salt used was the same (we emphasize on *at least*). We have one salt sampled every time a query is made to the signing oracle, and it can collide each time with a previous salt or any of the queries to the Hash_i. This means, we can bound this with:

$$|\mathsf{Pr}_2[\mathsf{Forge}] - \mathsf{Pr}_3[\mathsf{Forge}]| \leq \frac{q_S \cdot (q_S + q_1 + q_2 + q_3)}{2^{2\lambda}} \quad \leq \frac{q_S \cdot (q_S + 3 \cdot q')}{2^{2\lambda}}$$

– **Game 4.** When signing a message m, we now replace h_{sh} and h_{piop} with uniformly distributed random values. We then compute the challenges $\boldsymbol{\Gamma}$ and $\{i^{*(e)}\}_e$, and the value v_{grinding}, by expanding them. There is a difference with **Game 3** during a signing query, if a query to Hash_1 or Hash_2 was previously made. However, this does not happen as **Game 3** would already abort due to salt collision which means that:

$$\mathsf{Pr}_4[\mathsf{Forge}] = \mathsf{Pr}_3[\mathsf{Forge}]$$

– **Game 5.** We replace each $\mathsf{commit}_{e,i}$, and h_{com}, with a uniformly distributed random value. Since $h_{\text{com}} = \mathsf{Hash}_1(\{\mathsf{commit}_{e,i}\}_{e,i})$, there cannot be a collision on h_{com} as there is no collision on the commitments since $\mathsf{commit}_{e,i}$ is expanded using e and i. This then means that, to have a collision on two queries to Hash_3, the same salt must be used, which is already an invalid forgery from the previous games thus:

$$\mathsf{Pr}_5[\mathsf{Forge}] = \mathsf{Pr}_4[\mathsf{Forge}]$$

– **Game 6.** We now use the HVZK simulator of the proof of knowledge in order to generate the views of the parties randomly. $\mathcal{A}$ has an advantage of ϵ_{PRG} at most when generating the views (as it is his advantage to distinguish between a random and a true transcript). Hence, the difference with the previous game is given by:

$$|\mathsf{Pr}_6[\mathsf{Forge}] - \mathsf{Pr}_5[\mathsf{Forge}]| \leq \tau \cdot q_S \cdot \epsilon_{PRG}$$

– **Game 7.** Finally, we say that an execution with index (e^*, ctr) of a query $h_{\text{piop}} = \mathsf{Hash}_2\left(\mathsf{pk}, \mathsf{salt}, \mathsf{msg}, h_{\text{sh}}, (\boldsymbol{\alpha}^{(e)}_{\mathsf{mid}}, \boldsymbol{\alpha}^{(e)}_{\mathsf{base}})_{e\in[1,\dots,\tau]}\right)$ allows to retrieve a correct witness if:

- h_{com} is a query to Hash_3, i.e., $h_{\text{com}} = \mathsf{Hash}_3(\{\mathsf{commit}_{e,i}\}_{e,i})$;
- Each $\mathsf{commit}_{e,i}$ is a query to Hash_3, i.e., $\mathsf{commit}_{e,i} = \mathsf{Hash}_3(\mathsf{seed}_{e,i}, i \cdot \tau + e, \mathsf{salt})$
- h_{sh} is the output of a query to Hash_1, i.e., $h_{\text{sh}} = \mathsf{Hash}_1(h_{\mathsf{com}}, (\boldsymbol{S}^{(e)}_{\mathsf{aux}}, \boldsymbol{C'}^{(e)}_{\mathsf{aux}})_{e\in[1,\dots,\tau]})$;
- The matrices $\boldsymbol{S}, \boldsymbol{C'}$ defined by $\{\mathsf{state}_i^{(e^*)}\}_{i\in[1,\dots,N]}$ form a correct Syndrome MinRank solution;
- $v_{\text{grinding}} = \mathbf{0} \in \{0,1\}^w$ and $\pi_{\mathsf{BAVC}} \neq \perp$.

In such cases, one is able to retrieve the correct witness from $\{\mathsf{seed}_i^{(e^*)}\}_{i\in[1,\dots,N]}$, and as a consequence is able to solve the Syndrome MinRank instance which means that:

$$\mathsf{Pr}_7[\mathsf{Solve}] \leq \epsilon_{SMR}$$

Finally, we need to look at the upper bound of $\left|\mathsf{Pr}_7[\mathsf{Forge} \cap \overline{\mathsf{Solve}}]\right|$. Solve does not happen here, meaning that, to have a forgery after a query to Hash_2, $\mathcal{A}$ has no choice but to cheat either on $\boldsymbol{\Gamma}$ or on $\{i^{*(e^*)}\}$. For that, he can:

- Find matrices $\tilde{\boldsymbol{S}}$ and $\tilde{\boldsymbol{C}}'$ such that $\boldsymbol{H}\,\mathrm{vec}\left(\tilde{\boldsymbol{S}}\cdot\left[\boldsymbol{I}_r \parallel \tilde{\boldsymbol{C}}'\right]\right) \neq \boldsymbol{y}$ but such that $\boldsymbol{\Gamma}\cdot\left(\boldsymbol{H}\,\mathrm{vec}\left(\tilde{\boldsymbol{S}}\cdot\left[\boldsymbol{I}_r \parallel \tilde{\boldsymbol{C}}'\right]\right)-\boldsymbol{y}\right)=\boldsymbol{0}$, which happens with probability $p=\frac{1}{q^{\mu\cdot\rho}}$;

- Successfully cheat on the polynomial P_α, which happens with probability $\frac{2}{N}$ because there are 2 roots to this polynomial.

Cheating on the second round must happen on the τ repetitions thus the cheating probability is bounded by $q'\cdot\left(\frac{2}{N}\right)^\tau + q'\cdot\tau\cdot\frac{1}{q^{\mu\cdot\rho}}$. In addition, because v_{grinding} must be equal to $\boldsymbol{0}$, the adversary $\mathcal{A}$ has a success probability 2^{-w} for each iteration (e^*, ctr). This results in a probability to cheat bounded by $2^{-w}\cdot q'\cdot\left(\frac{2}{N}\right)^\tau + q'\cdot\tau\cdot\frac{1}{q^{\mu\cdot\rho}}$. Finally, the adversary must have $\pi_{\mathsf{BAVC}} \neq \perp$. Let θ be the probability that $\pi_{\mathsf{BAVC}} = \perp$ namely the probability that the sibling path exceeds the threshold T_{open}. This reduces the number of possible challenges from N^τ to $(N^\tau)\cdot(1-\theta)$. Thus, the adversary only has to guess among the $(N^\tau)\cdot(1-\theta)$ challenges which does not fail, but since the set $\{i^{*(e)}\}_e$ is uniformly sampled, the forgery will fail with probability θ. As a result, the adversary can cheat with probability $\frac{2^\tau}{(N^\tau)\cdot(1-\theta)}\cdot(1-\theta) = \left(\frac{2}{N}\right)^\tau$.

Computing the sum of the aforementioned upper bounds concludes the proof.

8 Known Attacks

8.1 Generic Attacks against Fiat–Shamir Signatures

It is possible to forge a signature without solving the underlying instance of the MinRank problem. For signature schemes built by applying the Fiat–Shamir transformation on a five-pass identification, Kales and Zaverucha proposed in [32] a forgery achieved by guessing separately the two challenge of the protocol. It results in an additive cost rather than the expected multiplicative cost. The cost associated with forging a transcript that passes the first 5 rounds of the Proof of Knowledge relies on achieving an optimal trade-off between the work needed for passing the first step and the work needed for passing the second step. To achieve the attack, one can find an optimal number of repetitions with the formula:

$$\tau' = \arg \min_{0 \leq \tau' \leq \tau} \left\{ \frac{1}{P_1} + \left(\frac{1}{P_2} \right)^{\tau - \tau'} \right\}$$

where P_1 and P_2 are the probabilities to pass respectively the first τ challenge τ' times and the second challenge $\tau - \tau'$ times.

In our case, P_1 corresponds to the probability of having a false-positive in the polynomial constraints protocol ρ times, and P_2 corresponds to the probability of cheating on the polynomial P_α (see Section 3.1). Therefore, the KZ attack complexity is given by

$$\text{cost}_{\text{forge}} = \min_{0 \leq \tau' \leq \tau} \left\{ \frac{1}{\sum_{i=\tau'}^{\tau} \binom{\tau}{i} p^i (1-p)^{\tau - i}} + (\frac{N}{2})^{\tau - \tau'} \right\}$$

where $p = \frac{1}{q^{\mu \cdot \rho}}$.

8.2 Known attacks against the MinRank Problem

There are two types of attacks on the MinRank problem: combinatorial and algebraic. In the following, we detail the most efficient approaches of both categories, as well as a hybrid strategy. In this section, all complexity formulas are given in terms of multiplications in $\mathbb{F}_q$, and $\omega \in [2, 3]$ denotes the constant of matrix multiplication, that is, two $n \times n$ matrices over a field can be multiplied in $\mathcal{O}(n^\omega)$ field multiplications.

Hybrid strategy. We combine two hybridization strategies to solve a given instance of the MinRank. The first one guesses l_v coefficients of the solution vector $(\alpha_1, \ldots, \alpha_k)$. The second one, introduced by Bardet et al. [6], guesses a vectors (with a specific structure) in the right kernel of the secret $\boldsymbol{E}$. For each guess of the combined approach one derives a MinRank instance $\tilde{\boldsymbol{M}}_0, \ldots, \tilde{\boldsymbol{M}}_k$ with parameters $(q, m, n-a, k-am-l_v, r)$. Hence the complexity of this hybrid

approach is given by

$$q^{a\cdot r+l_v}\Big(\mathsf{MR\_Complexity}(q,m,n-a,k-am-l_v,r) + \big(\min\{k,am\}\big)^{\omega}\Big), \tag{4}$$

where $\mathsf{MR\_Complexity}(\cdot)$ gives the complexity to solve a random instance of the MinRank problem defined by the input parameters.

Combinatorial attacks. We describe below attacks based on enumerating coefficients, enumerating entries of $\boldsymbol{E}$ and Kernel search.

Enumerating coefficients. The most trivial way to solve the MinRank problem is to try all possible solution vectors in $\mathbb{F}_q^k$. For each vector, we compute the corresponding linear combination with the input matrices $\boldsymbol{M}_0,\ldots,\boldsymbol{M}_k \in \mathbb{F}_q^{m\times n}$ and check if it has rank at most r. The complexity of this attack is $\mathcal{O}\big(q^k\cdot r^\omega\big)$ [14].

Enumerating entries of $\boldsymbol{E}$. Another way is to enumerate some entries of the hidden matrix $\boldsymbol{E}$. This attack is particularly effective when $(m-r)(n-r)<k$, and it is known as the big-k attack[3].

Kernel-search. The kernel-search algorithm was introduced by Goubin and Courtois [28]. It consists of guessing $\lceil k/m\rceil$ linearly independent vectors in the kernel of the unknown rank r matrix $\boldsymbol{E}$. The expected complexity of this algorithm is

$$\mathcal{O}\left(q^{r\cdot\lceil k/m\rceil}\cdot k^\omega\right).$$

Algebraic attacks. Let $\boldsymbol{M}_0,\ldots,\boldsymbol{M}_k \in \mathbb{F}_q^{m\times n}$ be an instance of the MinRank problem with target rank r. Let $\boldsymbol{M}$ be the formal matrix of linear forms in the α_i's defined by $\boldsymbol{M}=\boldsymbol{M}_0+\sum_{i=1}^k \alpha_i\boldsymbol{M}_i$.

Kipnis-Shamir modeling. The first algebraic modeling to solve the MinRank problem was proposed in 1999 by Kipnis and Shamir [33]. It is based on the following fact: If there is a vector $(\alpha_1,\ldots,\alpha_k)\in\mathbb{F}_q^k$ and a matrix $\boldsymbol{K}\in\mathbb{F}_q^{r\times(n-r)}$ such that

$$\boldsymbol{M}\cdot\begin{bmatrix}\boldsymbol{I}_{n-r}\\ -\boldsymbol{K}\end{bmatrix}=\boldsymbol{0}_{m\times(n-r)}, \tag{5}$$

where $\boldsymbol{I}_{n-r}\in\mathbb{F}_q^{(n-r)\times(n-r)}$ is a non-singular matrix, then the vector $(\alpha_1,\ldots,\alpha_k)$ is a solution to the instance $\boldsymbol{M}_0,\ldots,\boldsymbol{M}_k\in\mathbb{F}_q^{m\times n}$.

The Kipnis-Shamir modeling consists on solving with algebraic techniques the system coming from the entries of the matrix equation in Eq. (5). The unknowns

[3] Originally called the big-m attack, since m was the variable used to denote the number of matrices in the MinRank problem [14].

being the α_i's and the entries of the matrix $\boldsymbol{K}$. However, its complexity is not well understood, in particular because, according to [5,29], the Gröbner basis computation will produce the Minors and Support-Minors equations that we describe below. It is then more interesting from the computational point of view to directly consider the Minors or Support-Minors modelings

Minors modeling. This modeling was presented and analyzed in [19,20]. The algebraic system consists of all the minors of size $r+1$ of $\boldsymbol{M}$. This is a determinantal system, whose Hilbert series is given by

$$HS(t) := \left[(1-t)^{(m-r)(n-r)-(k+1)} \frac{\det(A(t))}{t^{\binom{r}{2}}}\right],$$
$$\text{with } A(t) = \left(\sum_{\ell=0}^{\max(m-i,n-j)} \binom{m-i}{\ell}\binom{n-j}{\ell} t^\ell\right)_{1\le i\le r, 1\le j\le r}$$

where for a series $S \in \mathbb{Z}[[t]]$, $[S]$ denotes the series obtained by truncating S at the first non-positive coefficient.

As long as $k < (m-r)(n-r)$, the series is a polynomial, and the degree of regularity D of the system is $\deg(HS)+1$. The complexity of the Gröbner basis computation can then be estimated as the cost of computing the echelon form on the Macaulay matrix of the system in degree D that has $\binom{k+D}{D}$ columns and almost the same rank. As shown in [27], it is possible to refine the F5 algorithm to construct subs-matrices of the Macaulay matrices with a number of rows equal to its rank. Even if this algorithm has not been designed yet for the Minors modeling for any parameters set, we estimate the complexity of computing the echelon form as

$$O\left(\binom{k+D}{D}^\omega\right), \quad D = \deg(HS(z)) + 1.$$

Support-Minors Modeling. The Support-Minors modeling was introduced in [7]. The idea is to consider the vector space generated by the rows of a column submatrix $\boldsymbol{M'}$ of $\boldsymbol{M}$, that has rank r and $n' \ge r+1$ columns, to introduce a formal matrix $\boldsymbol{C} \in \mathbb{F}_q^{r\times n'}$ representing a generator matrix of this vector space. As $\boldsymbol{C}$ is a basis for the rows of $\boldsymbol{M'}$, each row m_i of $\boldsymbol{M'}$ is a linear combination of the rows of $\boldsymbol{C}$. This leads to the algebraic system

$$\mathsf{SupportMinors} = \left\{\mathsf{MaximalMinors}\left(\binom{m_i}{\boldsymbol{C}}\right) : m_i \text{ row of } \boldsymbol{M'}\right\}.$$

Considering the maximal minors of $\boldsymbol{C}$ as new variables, the system consists of bilinear equations in the α_i's and the minors of $\boldsymbol{C}$. It is conjectured in [7], based on a theoretical analysis of the system and some experimental heuristics, that it is possible to solve this bilinear system by linear algebra on the Macaulay matrix at augmented degree $b \le \min(q-1, r+1, n'-r)$ in the α_i's and degree one in

the minors of $\boldsymbol{C}$. This matrix has rank N_b and M_b columns with

$$N_b = \begin{cases} \sum_{i=1}^{b} (-1)^{i+1} \binom{n'}{r+i} \binom{m+i-1}{i} \binom{k+b-i-1}{b-i} & \text{if } q > 2 \\ \sum_{j=1}^{b} \sum_{i=1}^{j} (-1)^{i+1} \binom{n'}{r+i} \binom{\eta+i-1}{i} \binom{m}{j-i} & \text{if } q = 2. \end{cases}$$

$$M_b = \begin{cases} \binom{k+b-1}{b} \binom{n'}{r} & \text{if } q > 2 \\ \sum_{j=1}^{b} \binom{n'}{r} \binom{m}{j} & \text{if } q = 2. \end{cases}$$

The degree b is computed as the smallest degree such that $N_b \geq M_b - 1$. In this case, the complexity of the Support-Minors modeling is given by

$$\min \left\{ 3 \cdot k(r+1) \cdot M_b^2,\ 7 \cdot N_b {M_b}^{\omega-1} \right\}.$$

Quantum attacks. We use the quantum algorithm introduced in [1]. This algorithm executes a Grover search of some of the columns of the secret matrix $\boldsymbol{K}$ from the Kipnis-Shamir modeling (see Eq. (5)).

Let $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_{n-r} \in \mathbb{F}_q^r$ denote the columns $\boldsymbol{K}$, and let $t > (k+1)/m$ be an integer. Define $B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t) \in \mathbb{F}_q^{(k+1)\times mt}$ as the matrix such that $(\alpha_1, \ldots, \alpha_k, 1) \cdot B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t)$ gives the bilinear equations (in the α_i's and the entries of the $\boldsymbol{k}_j$'s) in the first t columns of $\boldsymbol{M} \cdot \left[\boldsymbol{I}_{n-r} \ -\boldsymbol{K}^\top\right]^\top$ in a fixed order.

By the Kipnis-Shamir modeling, it holds that

$$(\alpha_1, \ldots, \alpha_k, 1) \cdot B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t) = \boldsymbol{0}_{1\times mt}.$$

Therefore, the rank of $B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t)$ is less than $k+1$.

Let $\mathcal{S}$ be the of all possible q^{rt} matrices of the form $B(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t) \in \mathbb{F}_q^{(k+1)\times mt}$, with $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t \in \mathbb{F}_q^r$. For the algorithm to work, we need to execute it using a value t such that $B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t)$ is the only matrix in $\mathcal{S}$ with rank less than $k+1$. For such a value t, the quantum complexity of this attack is

$$\mathcal{O}\big(2^{(rt \log q)/2} \cdot (k+1)^2 mt\big)$$

operations over $\mathbb{F}_q$.

Determing the minimum t such that $B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t)$ is the only matrix in $\mathcal{S}$ with rank less than $k+1$ plays a key role in the complexity of the attack. Unlike [1], we assume that a randomly selected matrix from $\mathcal{S}$ is of rank less than $k+1$ with probability $\mathcal{O}\left(q^{-(mt-k)}\right)$, which is the probability that a random matrix in $\mathbb{F}_q^{(k+1)\times mt}$ is not full rank. Therefore, as long as $k/(m-r) < t$, we expect that $B(\boldsymbol{k}_1, \ldots, \boldsymbol{k}_t)$ is the only matrix in $\mathcal{S}$ with rank less than $k+1$. However, the choice $t = \lceil (k+1)/m \rceil$ gives a lower-bound of the quantum complexity of the algorithm. This is the value used to compute the margins in Table 13.

Limited circuit depth. For the quantum security definition of NIST categories 1, 3 and 5, the maximum depth of the used quantum circuits is limited to $2^{\texttt{maxdepth}}$ with $\texttt{maxdepth} \leq 96$. A parameter set is said to match the quantum security definition for a category if an attack requires at least $2^{b-\texttt{maxdepth}}$ quantum gates for $b = 157, 221, 285$ for category 1, 3 and 5, respectively.

We lower bound the depth of the described quantum circuit by

$$D = 2^{(rt \log q)/2} k^2,$$

which corresponds to the sequential repetition of the Grover iterations, where we lower bound the depth of the oracle with k^2.

In the case of $D > 2^{\texttt{maxdepth}}$, the most efficient strategy to restrict the depth of the quantum circuit is to partition the search space in P equally sized sets [34]. Then the search has to be reapplied for each partition, which comes at a depth of

$$D_P = \frac{D}{\sqrt{P}}.$$

and leads to $D_P = 2^{\texttt{maxdepth}}$ for a choice of $P = (D/2^{\texttt{maxdepth}})^2$.

The total number of quantum gates necessary to launch the depth-limited attack becomes

$$T = \mathcal{O}(P \cdot D_P \cdot mt \log^2 q), \tag{6}$$

where we count $(\log q)^2$ gates per field multiplication.

In Table 13, we state the respective parameter t for each parameter set and the security margin. A margin of x implies that a quantum attack on the respective parameter set is 2^x times as costly as a quantum attack on AES with corresponding key length (using the NIST metric). Since all margins are found to be positive, quantum attacks on Mirath remain less efficient than those on AES.

q	Category	t	margin
2	NIST-1	35	36
	NIST-3	41	39
	NIST-5	45	41
16	NIST-1	9	27
	NIST-3	11	41
	NIST-5	12	47

Table 13. Quantum security margin of different parameter sets.

9 Advantages and Limitations

One of the key aspects of Mirath is that it is constructed from a zero-knowledge proof of knowledge via the Fiat–Shamir transform. This approach inherits many advantages, but it also bears some limitations.

9.1 Advantages

Security. The main hardness assumption of Mirath is the difficulty of solving random *non-structured* instances of the MinRank problem. This well-established NP-complete problem has been significantly studied in the cryptanalysis of multivariate-based cryptosystems and in the construction of the post-quantum signature schemes MiRitH and MIRA. Furthermore, due to its many applications in cryptography and cryptanalysis, the hardness of the MinRank problem is well-understood by the cryptographic community. Hence, it constitutes a conservative assumption.

Signature + public key size. Mirath offers competitive signature sizes using very small public keys, which yields a competitive signature + public key size. For NIST security level 1, the sum of the signature and public key sizes of Mirath gives 2.9 kB for ($q = 2$) and 3.1 kB for ($q = 16$), which are both smaller than the post-quantum NIST standards ML-DSA (Dillitium) and SLH-DSA (SPHINCS+) with 3.7 kB and 7.8kB respectively.

Resilience against MinRank attacks: Because of the way the size of the signature is obtained: one part is related to the MPC and only dependent on the security level and one part is related to the parameters of the problem in themselves, increasing the size of the problem parameters has a limited impact on the total size of the signature. Thus, if one later discovers effective attacks against the MinRank problem that force us to increase the parameters, this would have a limited impact on the scheme.

Extension to ring signatures. A *ring signature scheme* enables a user to sign a message so that a *ring* of possible signers (of which the user is a member) is identified without revealing exactly which member of that ring actually generated the signature [11]. Mirath can be easily extended to a ring signature scheme as shown in [10, Section 5] (as any MinRank-based scheme), or more recently as shown in [24, Section 6.6] (as any MPCitH-based scheme). We would obtain signature sizes smaller than 4 KB for a small number of users and smaller than 5 KB for a large number of users.

9.2 Limitations

Growth rate of the signature size. The signature size almost doubles when increasing the security level. This comes from the fact that both the MinRank instance and the number of repetitions need to increase linearly, since both the

Fiat-Shamir transform and the MinRank instance need to reach much higher bit security.

Efficiency. MPCitH frameworks involves numerous call to symmetric primitives which makes Mirath slower than the NIST post-quantum standard ML-DSA. Besides this, in general, the efficiency of Mirath is competitive when compared with other post-quantum signature schemes.

Low-cost devices and embedded systems. Mirath might be particularly heavy for low-cost devices such as smart cards or embedded systems, although it has the potential to perform well on hardware as being highly parallelizable.

A Variant using the VOLEitH Framework

In this section, we provide key and signature sizes for Mirath-v, a variant of Mirath using the VOLEitH framework as described in Section 3.1.

Instance	sk size [B]	pk size [B]	σ size [B]
Mirath-v-1a-short	32	73	2,998
Mirath-v-1a-fast	32	73	3,640
Mirath-v-3a-short	48	107	6,800
Mirath-v-3a-fast	48	107	8,244
Mirath-v-5a-short	64	147	12,368
Mirath-v-5a-fast	64	147	14,744

Table 14. Keys and signature sizes of Mirath-v-a (VOLE variant, $q = 16$)

Instance	sk size [B]	pk size [B]	σ size [B]
Mirath-v-1b-short	32	57	2,822
Mirath-v-1b-fast	32	57	3,384
Mirath-v-3b-short	48	84	6,430
Mirath-v-3b-fast	48	84	7,689
Mirath-v-5b-short	64	112	11,609
Mirath-v-5b-fast	64	112	13,640

Table 15. Keys and signature sizes of Mirath-v-b (VOLE variant, $q = 2$)

B Variant with Smaller Public Keys

We propose a second key-generation algorithm for Mirath, which we call KeyGen2. It has the advantage of producing smaller public keys, but the disadvantage of a slower key-generation. KeyGen2 is given as Algorithm 26, and the corresponding algorithms for key decompression are Algorithms 27 and 28, which in turn rely on the three subroutines described by Algorithms 29, 30, and 31.

Algorithm 26 is based on the method of Di Scala and Sanna [16] and generates a public key of the form $\mathsf{pk} = (\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}_B)$, where $\mathsf{seed}_{\mathsf{pk}}$ is a seed to generated $\boldsymbol{H}'$ and $\boldsymbol{y}_A$. Hence, the size of the public key is

$$|\mathsf{pk}| = |\mathsf{seed}_{\mathsf{pk}}| + |\boldsymbol{y}_B| = \lambda + \big(m(n-r) - k\big) \log q$$

bits. A comparison of the public-key size of KeyGen and KeyGen2 for the parameter sets of Mirath is provided in Table 16.

	public-key size [B]	
Instance	KeyGen	KeyGen2
Mirath-1a	73	41
Mirath-3a	107	60
Mirath-5a	147	81
Mirath-1b	57	36
Mirath-3b	84	53
Mirath-5b	112	70

Table 16. Public-key sizes of KeyGen and KeyGen2.

Algorithm 26 Routine KeyGen2.

Mirath.KeyGen2()

for t from 1 to max_tries **do**
  $\mathsf{seed}_{\mathsf{pk}} \leftarrow \mathsf{TRG.Seed}()$
  $\mathsf{seed}_{\mathsf{sk}} \leftarrow \mathsf{TRG.Seed}()$
  $(\boldsymbol{M}_A[\,], \boldsymbol{H}_B[\,]) \leftarrow \mathsf{ExpandSeedMAHB}(\mathsf{seed}_{\mathsf{pk}})$
    $\triangleright$ $\boldsymbol{M}_A[i] \in \mathbb{F}_q^{m \times r}$ $(i = 0, \ldots, k)$, $\boldsymbol{H}_B[j] \in \mathbb{F}_q^{m_0 \times 1}$ $(j = 1, \ldots, k)$
  $\boldsymbol{C}' \leftarrow \mathsf{ExpandSeedC}(\mathsf{seed}_{\mathsf{sk}})$ $\triangleright$ $\boldsymbol{C}' \in \mathbb{F}_q^{r \times (n-r)}$
  $\boldsymbol{\alpha} \leftarrow \mathsf{GetSpecialSolution}(\boldsymbol{M}_A[\,], \boldsymbol{C}')$ $\triangleright$ $\boldsymbol{\alpha} \in \mathbb{F}_q^{k \times 1}$ or $\boldsymbol{\alpha} = \perp$
  if $\boldsymbol{\alpha} \neq \perp$ **then**
    $\boldsymbol{S} \leftarrow \boldsymbol{M}_A[0] + \sum_{i=1}^{k} \alpha_i \boldsymbol{M}_A[i]$ $\triangleright$ $\boldsymbol{S} \in \mathbb{F}_q^{m \times r}$
    $\boldsymbol{H}_B[0] \leftarrow$ first m_0 entries of $\mathrm{vec}(\boldsymbol{S}\boldsymbol{C}')$ $\triangleright$ $\boldsymbol{H}_B[0] \in \mathbb{F}_q^{m_0 \times 1}$
    $\boldsymbol{y}_B \leftarrow \boldsymbol{H}_B[0] - \sum_{i=1}^{k} \alpha_i \boldsymbol{H}_A[i]$ $\triangleright$ $\boldsymbol{y}_B \in \mathbb{F}_q^{m_0 \times 1}$
    $\mathsf{pk} \leftarrow (\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}_B)$
    $\mathsf{sk} \leftarrow (\mathsf{seed}_{\mathsf{sk}}, \mathsf{seed}_{\mathsf{pk}})$
    return $(\mathsf{pk}, \mathsf{sk})$
  endif
endfor
return $\perp$

Algorithm 27 Routine DecompressPK2.

Mirath.DecompressPK2(pk)

$(\mathsf{seed}_{\mathsf{pk}}, \boldsymbol{y}_B) \leftarrow \mathsf{pk}$
$(\boldsymbol{M}_A[\,], \boldsymbol{H}_B[\,]) \leftarrow \mathsf{ExpandSeedMAHB}(\mathsf{seed}_{\mathsf{pk}})$
  $\triangleright$ $\boldsymbol{M}_A[i] \in \mathbb{F}_q^{m \times r}$ $(i = 0, \ldots, k)$, $\boldsymbol{H}_B[j] \in \mathbb{F}_q^{m_0 \times 1}$ $(j = 1, \ldots, k)$
$\boldsymbol{H}' \leftarrow -\begin{pmatrix} \mathrm{vec}(\boldsymbol{M}_A[1]) & \cdots & \mathrm{vec}(\boldsymbol{M}_A[k]) \\ \boldsymbol{H}_B[1] & \cdots & \boldsymbol{H}_B[1] \end{pmatrix}$ $\triangleright$ $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k) \times k}$
$\boldsymbol{y} \leftarrow \begin{pmatrix} \mathrm{vec}(\boldsymbol{M}_A[0]) \\ \boldsymbol{y}_B \end{pmatrix}$
return $(\boldsymbol{H}', \boldsymbol{y})$

Algorithm 28 Routine DecompressSK2.

Mirath.DecompressSK2(sk)

$\mathsf{seed}_{\mathsf{pk}}, \mathsf{seed}_{\mathsf{sk}} \leftarrow \mathsf{sk}$
$(\boldsymbol{M}_A[\,], \boldsymbol{H}_B[\,]) \leftarrow \mathsf{ExpandSeedMAHB}(\mathsf{seed}_{\mathsf{pk}})$
$\triangleright$ $\boldsymbol{M}_A[i] \in \mathbb{F}_q^{m \times r}$ $(i = 0, \ldots, k)$, $\boldsymbol{H}_B[j] \in \mathbb{F}_q^{m_0 \times 1}$ $(j = 1, \ldots, k)$
$\boldsymbol{C}' \leftarrow \mathsf{ExpandSeedC}(\mathsf{seed}_{\mathsf{sk}})$ $\triangleright$ $\boldsymbol{K} \in \mathbb{F}_q^{r \times (n-r)}$
$\boldsymbol{\alpha} \leftarrow \mathsf{GetSpecialSolution}(\boldsymbol{M}_A[\,], \boldsymbol{C}')$ $\triangleright$ $\boldsymbol{\alpha} \in \mathbb{F}_q^{k \times 1}$ or $\boldsymbol{\alpha} = \perp$
if $\boldsymbol{\alpha} = \perp$ **then return** $\perp$
$\boldsymbol{S} \leftarrow \boldsymbol{M}_A[0] + \sum_{i=1}^{k} \alpha_i \boldsymbol{M}_A[i]$
$\boldsymbol{H}_B[0] \leftarrow$ first m_0 entries of $\mathrm{vec}(\boldsymbol{S}\boldsymbol{C}')$ $\triangleright$ $\boldsymbol{H}_B[0] \in \mathbb{F}_q^{m_0 \times 1}$
$\boldsymbol{y}_B \leftarrow \boldsymbol{H}_B[0] - \sum_{i=1}^{k} \alpha_i \boldsymbol{H}_A[i]$ $\triangleright$ $\boldsymbol{y}_B \in \mathbb{F}_q^{m_0 \times 1}$
$\boldsymbol{H}' \leftarrow -\begin{pmatrix} \mathrm{vec}(\boldsymbol{M}_A[1]) & \cdots & \mathrm{vec}(\boldsymbol{M}_A[k]) \\ \boldsymbol{H}_B[1] & \cdots & \boldsymbol{H}_B[k] \end{pmatrix}$ $\triangleright$ $\boldsymbol{H}' \in \mathbb{F}_q^{(mn-k) \times k}$
$\boldsymbol{y} \leftarrow \begin{pmatrix} \mathrm{vec}(\boldsymbol{M}_A[0]) \\ \boldsymbol{y}_B \end{pmatrix}$
return $(\boldsymbol{H}', \boldsymbol{y}, \boldsymbol{S}, \boldsymbol{C}')$

Algorithm 29 Routine ExpandSeedMAHB.

Mirath.ExpandSeedMAHB($\mathsf{seed}_{\mathsf{pk}}$)

$\mathsf{prg} \leftarrow \mathsf{PRG.Init}(\mathsf{seed}_{\mathsf{pk}})$
for i from 0 to k **do**
  $\boldsymbol{M}_A[i] \leftarrow \mathsf{PRG.Sample}(\mathsf{prg}, \mathbb{F}_q^{m \times r})$
endfor
for i from 1 to k **do**
  $\boldsymbol{H}_B[i] \leftarrow \mathsf{PRG.Sample}(\mathsf{prg}, \mathbb{F}_q^{m_0 \times 1})$
endfor
return $(\boldsymbol{M}_A[\,], \boldsymbol{H}_B[\,])$

Algorithm 30 Routine ExpandSeedC.

Mirath.ExpandSeedC($\mathsf{seed}_{\mathsf{sk}}$)

$\mathsf{prg} \leftarrow \mathsf{PRG.Init}(\mathsf{seed}_{\mathsf{sk}})$
$\boldsymbol{C}' \leftarrow \mathsf{PRG.Sample}(\mathsf{prg}, \mathbb{F}_q^{r \times (n-r)})$
return $\boldsymbol{C}'$

Algorithm 31 Routine GetSpecialSolution.

GetSpecialSolution($\boldsymbol{M}_A[], \boldsymbol{C}'$)

1 : **for** i from 1 to k **do**
2 : **for** j from 0 to k **do**
3 : $a_{i,j} \leftarrow m_0 + i$ entry of $\mathrm{vec}(\boldsymbol{M}_{A,j}\boldsymbol{C}')$
▷ Note: no need to compute the full product $\boldsymbol{M}_{A,j}\boldsymbol{C}'$, since only one entry is needed
4 : **endfor**
5 : **endfor**
6 : $\boldsymbol{A} \leftarrow \boldsymbol{I} - (a_{i,j})_{1 \le i,j \le k}$ ▷ $\boldsymbol{A} \in \mathbb{F}_q^{k \times k}$
7 : $\boldsymbol{b} \leftarrow (a_{i,0})_{1 \le i \le k}$ ▷ $\boldsymbol{b} \in \mathbb{F}_q^{k \times 1}$
8 : Compute $\boldsymbol{\alpha} \in \mathbb{F}_q^{k \times 1}$ as the unique solution to the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, if such a solution exists unique, otherwise return $\perp$
9 : **return** $\boldsymbol{\alpha}$

References

1. G. Adj, S. Barbero, E. Bellini, A. Esser, L. Rivera-Zamarripa, C. Sanna, J. A. Verbel, and F. Zweydinger, *MiRitH: Efficient post-quantum signatures from MinRank in the head*, IACR TCHES **2024** (2024), no. 2, 304–328.
2. G. Adj, L. Rivera-Zamarripa, J. Verbel, E. Bellini, S. Barbero, A. Esser, C. Sanna, and F. Zweydinger, *MiRitH — MinRank in the Head*, Tech. report, National Institute of Standards and Technology, 2023, available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.
3. N. Aragon, M. Bardet, L. Bidoux, J. Chi-Domínguez, V. Dyseryn, T. Feneuil, P. Gaborit, R. Neveu, M. Rivain, and J. Tillich, *MIRA*, Tech. report, National Institute of Standards and Technology, 2023, available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.
4. N. Aragon, L. Bidoux, J.-J. Chi-Domínguez, T. Feneuil, P. Gaborit, R. Neveu, and M. Rivain, *MIRA: a Digital Signature Scheme based on the MinRank problem and the MPC-in-the-Head paradigm*, arXiv preprint, 2023, https://arxiv.org/abs/2307.08575.
5. M. Bardet and M. Bertin, *Improvement of algebraic attacks for solving superdetermined MinRank instances*, Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022 (J. H. Cheon and T. Johansson, eds.), Springer, Cham, September 2022, pp. 107–123.
6. M. Bardet, P. Briaud, M. Bros, P. Gaborit, and J.-P. Tillich, *Revisiting algebraic attacks on MinRank and on the rank decoding problem*, DCC **91** (2023), no. 11, 3671–3707.
7. M. Bardet, M. Bros, D. Cabarcas, P. Gaborit, R. A. Perlner, D. Smith-Tone, J.-P. Tillich, and J. A. Verbel, *Improvements of algebraic attacks for solving the rank decoding and MinRank problems*, ASIACRYPT 2020, Part I (S. Moriai and H. Wang, eds.), LNCS, vol. 12491, Springer, Cham, December 2020, pp. 507–536.
8. C. Baum, W. Beullens, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl, *One tree to rule them all: Optimizing ggm trees and owfs for post-quantum signatures*, Cryptology ePrint Archive, Paper 2024/490, 2024, https://eprint.iacr.org/2024/490.
9. C. Baum, L. Braun, C. Delpech de Saint Guilhem, M. Klooß, E. Orsini, L. Roy, and P. Scholl, *Publicly verifiable zero-knowledge and post-quantum signatures from VOLE-in-the-head*, CRYPTO 2023, Part V (H. Handschuh and A. Lysyanskaya, eds.), LNCS, vol. 14085, Springer, Cham, August 2023, pp. 581–615.
10. E. Bellini, A. Esser, C. Sanna, and J. A. Verbel, *MR-DSS - smaller MinRank-based (ring-)signatures*, Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022 (J. H. Cheon and T. Johansson, eds.), Springer, Cham, September 2022, pp. 144–169.
11. A. Bender, J. Katz, and R. Morselli, *Ring signatures: Stronger definitions, and constructions without random oracles*, J. Cryptology **22** (2009), no. 1, 114–138.
12. L. Bidoux, T. Feneuil, P. Gaborit, R. Neveu, and M. Rivain, *Dual support decomposition in the head: Shorter signatures from rank SD and MinRank*, Advances in Cryptology – ASIACRYPT 2024 (Singapore) (K.-M. Chung and Y. Sasaki, eds.), Springer Nature Singapore, 2024, pp. 38–69.
13. J. F. Buss, G. S. Frandsen, and J. O. Shallit, *The computational complexity of some problems of linear algebra*, STACS 97 (Berlin, Heidelberg) (R. Reischuk and M. Morvan, eds.), Springer Berlin Heidelberg, 1997, pp. 451–462.

14. N. Courtois, *Efficient zero-knowledge authentication based on a linear algebra problem MinRank*, ASIACRYPT 2001 (C. Boyd, ed.), LNCS, vol. 2248, Springer, Berlin, Heidelberg, December 2001, pp. 402–421.
15. R. Cramer, I. Damgård, and Y. Ishai, *Share conversion, pseudorandom secret-sharing and applications to secure computation*, TCC 2005 (J. Kilian, ed.), LNCS, vol. 3378, Springer, Berlin, Heidelberg, February 2005, pp. 342–362.
16. A. Di Scala and C. Sanna, *Smaller public keys for MinRank-based schemes*, Journal of Mathematical Cryptology **19** (2025), 20240008, https://doi.org/10.1515/jmc-2024-0008.
17. N. C. S. Division, *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, May 2014.
18. A. Esser, J. A. Verbel, F. Zweydinger, and E. Bellini, *SoK: CryptographicEstimators - a software library for cryptographic hardness estimation*, ASIACCS 24 (J. Zhou, T. Q. S. Quek, D. Gao, and A. A. Cárdenas, eds.), ACM Press, July 2024.
19. J.-C. Faugère, M. S. El Din, and P.-J. Spaenlehauer, *Computing loci of rank defects of linear matrices using gröbner bases and applications to cryptology*, Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation (New York, NY, USA), ISSAC '10, Association for Computing Machinery, 2010, p. 257–264.
20. J.-C. Faugère, M. S. El Din, and P.-J. Spaenlehauer, *On the complexity of the generalized minrank problem*, J. Symb. Comput. **55** (2013), 30–58.
21. T. Feneuil, *The Polynomial-IOP Vision of the Latest MPCitH Frameworks for Signature Schemes*, Post-Quantum Algebraic Cryptography - Workshop 2, Institut Henri Poincaré, Paris, France, 2024.
22. T. Feneuil and M. Rivain, *Threshold linear secret sharing to the rescue of MPC-in-the-head*, Cryptology ePrint Archive, Report 2022/1407, 2022.
23. T. Feneuil and M. Rivain, *Threshold computation in the head: Improved framework for post-quantum signatures and zero-knowledge arguments*, Cryptology ePrint Archive, Report 2023/1573, 2023.
24. T. Feneuil and M. Rivain, *Threshold Computation in the Head: Improved Framework for Post-Quantum Signatures and Zero-Knowledge Arguments*, Cryptology ePrint Archive, Paper 2023/1573, 2023.
25. T. Feneuil and M. Rivain, *Threshold linear secret sharing to the rescue of MPC-in-the-head*, ASIACRYPT 2023, Part I (J. Guo and R. Steinfeld, eds.), LNCS, vol. 14438, Springer, Singapore, December 2023, pp. 441–473.
26. A. Fiat and A. Shamir, *How to prove yourself: Practical solutions to identification and signature problems*, CRYPTO'86 (A. M. Odlyzko, ed.), LNCS, vol. 263, Springer, Berlin, Heidelberg, August 1987, pp. 186–194.
27. S. Gopalakrishnan, V. Neiger, and M. Safey El Din, *Refined F5 algorithms for ideals of minors of square matrices*, Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation (New York, NY, USA), ISSAC '23, Association for Computing Machinery, 2023, p. 270–279.
28. L. Goubin and N. Courtois, *Cryptanalysis of the TTM cryptosystem*, ASIACRYPT 2000 (T. Okamoto, ed.), LNCS, vol. 1976, Springer, Berlin, Heidelberg, December 2000, pp. 44–57.
29. H. Guo and J. Ding, *Algebraic relation of three minrank algebraic modelings*, Arithmetic of Finite Fields - 9th International Workshop, WAIFI 2022, Chengdu, China, August 29 - September 2, 2022, Revised Selected Papers (S. Mesnager and Z. Zhou, eds.), Lecture Notes in Computer Science, vol. 13638, Springer, 2022, pp. 239–249.

30. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, *Zero-knowledge from secure multiparty computation*, 39th ACM STOC (D. S. Johnson and U. Feige, eds.), ACM Press, June 2007, pp. 21–30.
31. M. Ito, A. Saito, and T. Nishizeki, *Secret sharing scheme realizing general access structure*, Electronics and Communications in Japan (Part III: Fundamental Electronic Science) **72** (1989), no. 9, 56–64.
32. D. Kales and G. Zaverucha, *An attack on some signature schemes constructed from five-pass identification schemes*, CANS 20 (S. Krenn, H. Shulman, and S. Vaudenay, eds.), LNCS, vol. 12579, Springer, Cham, December 2020, pp. 3–22.
33. A. Kipnis and A. Shamir, *Cryptanalysis of the HFE public key cryptosystem by relinearization*, CRYPTO'99 (M. J. Wiener, ed.), LNCS, vol. 1666, Springer, Berlin, Heidelberg, August 1999, pp. 19–30.
34. C. Zalka, *Grover's quantum searching algorithm is optimal*, Phys. Rev. A **60** (1999), 2746–2751.