

# Mirath Signature Scheme

23/09/2025

## 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)
- Thibault 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](mailto:team@pqc-mirath.org)

**Version:** 2.1.0

# Table of Contents

1	Introduction .....	6
2	Mathematical Background and Notations .....	7
2.1	Notations .....	7
2.2	MinRank Problem .....	7
2.3	All-but-one Vector Commitment (AVC) .....	8
3	High-Level Description of Mirath .....	10
3.1	TCitH Framework in the PIOP Formalism .....	10
3.2	Dual Support Modeling .....	14
3.3	Mirath Protocol .....	14
4	Detailed Description of Mirath .....	21
4.1	Notations .....	21
4.2	Finite Fields Representation .....	22
4.3	Matrices Representation .....	22
4.4	Hash Functions and Commitments .....	24
4.5	Randomness Generation and Sampling .....	24
4.6	Batched All-but-one Vector Commitments .....	28
4.7	Commitment to Polynomials .....	29
4.8	Computation of Polynomial Proof .....	32
4.9	Parsing of the Signature .....	33
4.10	Key Generation .....	34
4.11	Signing .....	36
4.12	Verification .....	37
5	Parameter Sets .....	38
5.1	MinRank Parameters .....	38
5.2	Protocol Parameters .....	39
5.3	Key and Signature Sizes .....	40
6	Performance Analysis .....	41
7	Security Analysis .....	44
8	Known Attacks .....	47
8.1	Generic Attacks against Fiat–Shamir Signatures. ....	47
8.2	Known attacks against the MinRank Problem. ....	47
9	Advantages and Limitations .....	52
9.1	Advantages .....	52
9.2	Limitations .....	52
A	Variant using the VOLEitH Framework .....	54
B	Variant with Smaller Public Keys .....	55

## List of Tables

1	Mathematical notation. . . . .	7
2	Algorithmic notation. . . . .	21
3	Polynomial modulus $f(x)$ for multiplications in $\mathbb{F}_{q^\mu}$ . . . . .	22
4	MinRank parameters used in Mirath-a ( $q = 16$ ) . . . . .	38
5	MinRank parameters used in Mirath-b ( $q = 2$ ) . . . . .	38
6	MPC parameters used in Mirath-a ( $q = 16$ ) . . . . .	39
7	MPC parameters used in Mirath-b ( $q = 2$ ) . . . . .	39
8	Keys and signature sizes of Mirath-a ( $q = 16$ ) . . . . .	40
9	Keys and signature sizes of Mirath-b ( $q = 2$ ) . . . . .	40
10	Performances of Mirath-a in Millions of CPU Cycles (Optimized implementation with Rijndael-based commitments) . . . . .	42
11	Performances of Mirath-a in Millions of CPU Cycles (Optimized implementation with SHA3-based commitments) . . . . .	42
12	Performances of Mirath-b in Millions of CPU Cycles (Optimized implementation with Rijndael-based commitments) . . . . .	43
13	Performances of Mirath-b in Millions of CPU Cycles (Optimized implementation with SHA3-based commitments) . . . . .	43
14	Quantum security margin of different parameter sets. . . . .	51
15	Keys and signature sizes of Mirath-v-a (VOLE variant, $q = 16$ ) . . . . .	54
16	Keys and signature sizes of Mirath-v-b (VOLE variant, $q = 2$ ) . . . . .	54
17	Public-key sizes of KeyGen and KeyGen2. . . . .	55

## List of Algorithms

1	Transform a byte array into a matrix in $\mathbb{F}^{n_r \times n_c}$ . . . . .	23
2	Children nodes generation. . . . .	25
3	Pseudorandom element in $\mathbb{F}_q^{m \times r} \times \mathbb{F}_q^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$ . . . . .	26
4	Routine <b>ExpandSeedPublicMatrix</b> . . . . .	26
5	Routine <b>ExpandSeedSecretMatrices</b> . . . . .	27
6	Expand challenge evaluation points. . . . .	27
7	Challenge matrix $\mathbf{I} \in \mathbb{F}_{q^\mu}^{(mn-k) \times k}$ . . . . .	28
8	Routine <b>BAVC.Commit</b> . . . . .	28
9	Routine <b>BAVC.Open</b> . . . . .	29
10	Routine <b>BAVC.Reconstruct</b> . . . . .	30
12	Open random evaluations. . . . .	30
11	Commit to witness $(P_S, P_{C'})$ and masking $P_v$ polynomials. . . . .	31
13	Compute evaluations of $P_S, P_{C'}$ and $P_v$ on the opened points. . . . .	32
14	Computation of the polynomial $P_\alpha$ . . . . .	33
15	Reconstruction of the polynomial $P_\alpha$ . . . . .	33
16	Unparse signature. . . . .	34
17	Parse signature. . . . .	34
18	Routine <b>ComputeY</b> . . . . .	35
19	Key generation algorithm. . . . .	35

20	Decompress public key routine. ....	35
21	Decompress secret key routine. ....	35
22	Signing algorithm. ....	36
23	Verification algorithm. ....	37
24	Routine KeyGen2. ....	56
25	Routine DecompressPK2. ....	56
26	Routine DecompressSK2. ....	57
27	Routine ExpandSeedMAHB. ....	57
28	Routine ExpandSeedC. ....	57
29	Routine GetSpecialSolution. ....	58

## Changelog

### Version 2.1.0 (23/09/2025)

- The implementations of Mirath have been improved (performance speed-up and bug fixing). The MPC parameters have been updated to adjust the performance/size trade-offs offered by the scheme (MinRank parameters remain unchanged).

### 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  $\mathbf{M}$ , we let  $\text{vec}(\mathbf{M})$  denotes the column-major vectorization of  $\mathbf{M}$ , that is, the  $n_r n_c \times 1$  column vector whose entries are the entries of  $\mathbf{M}$  in column-major order.

Symbol	Meaning
$\{0, 1\}^\ell$	Set of binary strings of length $\ell$ .
$\{0, 1\}^*$	Set of binary strings of finite length.
$\mathbf{str}_1 \parallel \mathbf{str}_2$	Concatenation of the binary strings $\mathbf{str}_1$ and $\mathbf{str}_2$ .
$\mathbb{F}_q$	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.
$\mathbf{I}_s$	The $s \times s$ identity matrix.
$\text{rank}(\mathbf{M})$	The rank of the matrix $\mathbf{M}$ .
$\text{vec}(\mathbf{M})$	columns-major vectorization of the matrix $\mathbf{M}$ .
$(\mathbf{A} \mid \mathbf{B})$	The matrix obtained by juxtaposing the matrices $\mathbf{A}$ and $\mathbf{B}$ .
$\log$	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  $\mathbf{M}_1, \dots, \mathbf{M}_k, \mathbf{E} \in \mathbb{F}_q^{m \times n}$  and  $\mathbf{x} := (x_1, \dots, x_k) \in \mathbb{F}_q^k$  be uniformly sampled such that*

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

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

Usually, the instance of the MinRank problem is generated by uniformly sampling the matrices  $\mathbf{M}, \mathbf{M}_1, \dots, \mathbf{M}_k$  so that  $\text{rank}\left(\mathbf{M} + \sum_{i=1}^k x_i \mathbf{M}_i\right) \leq r$ . 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  $\text{vec}$ . In fact, this application allows to obtain the generator matrix of the code  $\langle \mathbf{M}_1, \dots, \mathbf{M}_k \rangle$  as the matrix  $\mathbf{G} \in \mathbb{F}_q^{mn \times k}$  whose columns are  $\text{vec}(\mathbf{M}_1), \dots, \text{vec}(\mathbf{M}_k)$ , i.e.,

$$\mathbf{G} = (\text{vec}(\mathbf{M}_1) \dots \text{vec}(\mathbf{M}_k)) \ .$$

By using the dual matrix of the code generated by  $\mathbf{G}$ , i.e., using  $\mathbf{H} \in \mathbb{F}_q^{(mn-k) \times mn}$  such that  $\mathbf{H}\mathbf{G} = \mathbf{0}$ , 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  $\mathbf{H} := [\mathbf{I}_{mn-k} \ \mathbf{H}'] \in \mathbb{F}_q^{(mn-k) \times mn}$  where  $\mathbf{H}' \in \mathbb{F}_q^{(mn-k) \times k}$  is a uniformly sampled matrix, and  $\mathbf{y} \in \mathbb{F}_q^{mn-k}$ . The computational MinRank Syndrome Problem( $q, m, n, k, r$ ) asks to find  $\mathbf{E}$  such that*

$$\mathbf{H} \text{vec}(\mathbf{E}) = \mathbf{y} \quad \text{and} \quad \text{rank}(\mathbf{E}) \leq r.$$

### 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  $v_1, \dots, v_N$  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  $\lambda$ -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:*

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



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

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

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

$$((\text{seed}'_1, \dots, \text{seed}'_N), h'_{com}) \leftarrow \text{Reconstruct}(\pi_{AVC}, i^*).$$

The scheme is said binding if it is hard to produce two openings  $\pi_{AVC}$  and  $\pi'_{AVC}$  leading to the same commitment while having different messages. Finally, the scheme is said hiding if it is hard to recover  $\text{seed}_{i^*}$  from  $\pi_{AVC}$ .

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  $\pi_{AVC}$  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  $\tau$  AVC schemes in parallel. Therefore, instead of considering  $\tau$  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  $\tau$  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  $(N, \tau)$  is defined by the following algorithms:

- $\text{Commit}() \rightarrow (h_{com}, \text{key}, \{(\text{seed}_1^{(e)}, \dots, \text{seed}_N^{(e)})\}_{e \in [1, \tau]})$ : Output a commitment  $h_{com}$  with opening key  $\text{key}$  for messages  $(\text{seed}_1^{(e)}, \dots, \text{seed}_N^{(e)}) \in (\{0, 1\}^\lambda)^N$  for all  $e \in [1, \tau]$ .
- $\text{Open}(\text{key}, \{i^{*(e)}\}_e)$ : On an opening key  $\text{key}$  and  $\tau$  indexes  $i^{*(e)} \in \{1, \dots, N\}$ , output an opening  $\pi_{BAVC}$  for  $\{\text{seed}_i^{(e)}\}_{e, i \neq i^{*(e)}}$ .
- $\text{Reconstruct}(\pi_{BAVC}, \{i^{*(e)}\}_e) \rightarrow (\{\text{seed}_i^{(e)}\}_{e, i \neq i^{*(e)}}, h_{com})$ : On the opening  $\pi_{BAVC}$  and indexes  $i^{*(e)} \in \{1, \dots, N\}$ , output the messages  $\{\text{seed}_i^{(e)}\}_{e, i \neq i^{*(e)}}$ , 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  $\tau$  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<sup>1</sup>, 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  $P_j(X) = w_j \cdot X + (w_{\text{base}})_j$  for some  $(w_{\text{base}})_j \in \mathbb{F}_{q^\mu}$ . He also samples a random degree- $(d-1)$  polynomial  $P_0 \in \mathbb{F}_{q^\mu}[X]$ . He commits to those polynomials.

<sup>1</sup> 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  $\gamma_1, \dots, \gamma_m \in \mathbb{F}_{q^\mu}$  and sends them to the prover. The latter then reveals the degree- $(d-1)$  polynomial  $Q(X)$  defined as

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

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

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

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  $\pi$  that the evaluations are consistent with the commitment.
4. The verifier checks that the revealed evaluations are consistent with the commitment using  $\pi$  and checks that we have

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

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

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

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  $\gamma_1, \dots, \gamma_m$ , 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

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

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  $P_1, \dots, P_n$  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  $\rho$  times in parallel: the verifier chooses a random matrix  $\mathbf{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times m}$  and then the prover reveals  $\rho$  polynomials  $Q_1, \dots, Q_\rho$  such that

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

where  $P_{0,1}, \dots, P_{0,\rho}$  are  $\rho$  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  $\rho$  such that  $\rho \cdot \mu \cdot \log_2 q \geq \lambda$ , the soundness error is roughly  $d/N$ . When  $N$  is small (compared to  $2^\lambda$ ), 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  $\ell$  points, the prover needs to sample  $P_1, \dots, P_n$  as degree- $\ell$  polynomials to preserve zero-knowledge, and the soundness error would be

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

By taking  $\ell$  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  $\text{seed}_1, \dots, \text{seed}_N$ .
2. One expands each  $\text{seed}_i$  as  $w_{\text{rnd},i} := \text{PRG}(\text{seed}_i) \in \mathbb{F}_q^{\bar{n}}$  for  $i \in \{1, \dots, N\}$ , where  $\text{PRG}$  is a pseudorandom generator.
3. One computes

$$w_{\text{acc}} \leftarrow \sum_{i=1}^N w_{\text{rnd},i} \in \mathbb{F}_q^{\bar{n}}$$

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

where  $\phi : \{1, \dots, N\} \rightarrow \mathbb{F}_{q^\mu}$  is a public one-to-one function.

4. One reveals the auxiliary value  $\text{aux} := w - w_{\text{acc}}$ .
5. One defines  $P_j$  as

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

for all  $j$ .

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

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

Indeed, we have that

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

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  $\lambda$ -bit security we need to repeat the protocol  $\tau$  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  $\tau$  sets of polynomials  $\{P_i^{(1)}\}_i, \dots, \{P_i^{(\tau)}\}_i$  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  $N^\tau$  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  $\mathbf{E}$  is solution of the constraints

$$\begin{cases} \mathbf{H} \text{vec}(\mathbf{E}) = \mathbf{y}, \\ \text{rank}(\mathbf{E}) \leq r \end{cases}$$

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

$$\mathbf{H} \text{vec}(\mathbf{S}\mathbf{C}) = \mathbf{y} \quad \text{with} \quad \mathbf{C} = [\mathbf{I}_r \ \mathbf{C}'].$$

### 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  $P_S = \mathbf{S} \cdot X + \mathbf{S}_{\text{base}}$  and  $P_{C'} = \mathbf{C}' \cdot X + \mathbf{C}'_{\text{base}}$ , where  $\mathbf{S}_{\text{base}}$  and  $\mathbf{C}'_{\text{base}}$  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  $P_v = \mathbf{v} \cdot X + \mathbf{v}_{\text{base}}$  where  $\mathbf{v} \in \mathbb{F}_{q^\mu}^\rho$  and  $\mathbf{v}_{\text{base}} \in \mathbb{F}_{q^\mu}^\rho$ . He commits to those polynomials.
2. The verifier chooses a random matrix  $\mathbf{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$  and sends it to the prover. The latter reveals the degree-1 polynomial  $P_\alpha(X)$  defined as

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

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

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  $\mathbf{S}_{\text{eval}} := P_S(r)$ ,  $\mathbf{C}'_{\text{eval}} := P_{C'}(r)$  and  $\mathbf{v}_{\text{eval}} := P_v(r)$ , together with a proof  $\pi$  that the evaluations are consistent with the commitment.
4. The verifier checks that the revealed evaluations are consistent with the commitment using  $\pi$  and check that we have

$$P_\alpha(r) \stackrel{?}{=} \mathbf{v}_{\text{eval}} + \mathbf{I} \cdot (\mathbf{H} \cdot \text{vec}(\mathbf{E}_{\text{eval}}) - \mathbf{y} \cdot r^2) \in \mathbb{F}_{q^\mu}^\rho$$

with

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

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

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

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  $\text{seed}_1, \dots, \text{seed}_N$ .
2. One expands each  $\text{seed}_i$  as

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

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

3. One computes

$$\begin{aligned} (\mathbf{S}_{\text{acc}}, \mathbf{C}'_{\text{acc}}, \mathbf{v}_{\text{acc}}) &\leftarrow \left( \sum_{i=1}^N \mathbf{S}_{\text{rnd},i}, \sum_{i=1}^N \mathbf{C}'_{\text{rnd},i}, \sum_{i=1}^N \mathbf{v}_{\text{rnd},i} \right) \\ (\mathbf{S}_{\text{base}}, \mathbf{C}'_{\text{base}}, \mathbf{v}_{\text{base}}) &\leftarrow \left( - \sum_{i=1}^N \phi(i) \cdot \mathbf{S}_{\text{rnd},i}, - \sum_{i=1}^N \phi(i) \cdot \mathbf{C}'_{\text{rnd},i}, - \sum_{i=1}^N \phi(i) \cdot \mathbf{v}_{\text{rnd},i} \right) \end{aligned}$$

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

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

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

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

$$\begin{aligned}\mathbf{S}_{\text{eval}} &= \phi(i^*) \cdot \mathbf{S}_{\text{aux}} + \sum_{i=1, i \neq i^*}^N (\phi(i^*) - \phi(i)) \cdot \mathbf{S}_{\text{rnd}, i} \\ \mathbf{C}'_{\text{eval}} &= \phi(i^*) \cdot \mathbf{C}'_{\text{aux}} + \sum_{i=1, i \neq i^*}^N (\phi(i^*) - \phi(i)) \cdot \mathbf{C}'_{\text{rnd}, i} \\ \mathbf{v}_{\text{eval}} &= \sum_{i=1, i \neq i^*}^N (\phi(i^*) - \phi(i)) \cdot \mathbf{v}_{\text{rnd}, i}\end{aligned}$$

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

*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  $\tau$  times in parallel. It implies that we use  $\tau$  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  $\tau$  independent AVC schemes. In *Mirath*, we use the BAVC scheme described in [8], which proposes the “one-tree” optimization. Instead of considering  $\tau$  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  $i^{\text{th}}$  seed of the  $e^{\text{th}}$  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  $\tau$  leaves of the big tree is more efficient than opening all but one leaf in each of the  $\tau$  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  $T_{\text{open}}$ , 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  $T_{\text{open}}$ . 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  $\approx 0.2$ . The selected value of  $T_{\text{open}}$  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  $\theta$  hash computations for the opening challenge (instead of 1). While this strategy decreases the challenge space by a factor  $\theta$ , 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  $\theta$ ). As noticed by the authors of [8], this strategy can be thought of as losing  $\log_2 \theta$  bit of security (because of a smaller challenge space) which are regained thanks to a proof-of-work (performing an average of  $\theta$  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  $P_\alpha$  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  $P_E := \mathbf{E} \cdot X^2 + \mathbf{E}_{\text{mid}} \cdot X + \mathbf{E}_{\text{base}}$  and  $P_\alpha := \boldsymbol{\alpha} \cdot X^2 + \boldsymbol{\alpha}_{\text{mid}} \cdot X + \boldsymbol{\alpha}_{\text{base}}$ . The prover should compute  $P_E(X) := P_S(X) \cdot [P_{I_r}(X) \parallel P_{C'}]$ , meaning that he should compute

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

Then, the prover should compute the polynomial

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

meaning that he should compute

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

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

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

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

together with

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

**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  $\Gamma$  involved in the definition of  $P_\alpha$ , 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  $\tau$  times in parallel, the scheme is affected by the attack of Kales and Zaverucha [32]. To avoid incrementing the number  $\tau$  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  $v_{\text{grinding}}$  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  $T_{\text{open}}$ . 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  $(N, \tau)$  for the large tree, namely parameters achieving  $\lambda - w$  bits of security instead of  $\lambda$ . More precisely, the parameters  $N$ ,  $\tau$  and  $w$  will be chosen such that  $(2/N)^\tau \cdot 2^{-w} \leq 2^{-\lambda}$  to achieve a  $\lambda$ -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  $S_{\text{eval}}$ ,  $C'_{\text{eval}}$  and  $v_{\text{eval}}$ , the signature includes information (data enabling to derive  $\{\text{seed}_i\}_{i \neq i^*}$  and the auxiliary value  $(S_{\text{aux}}, C'_{\text{aux}})$ ) that enables the verifier to derive them.
- Instead of including the polynomial  $P_\alpha := \alpha_{\text{mid}} \cdot X + \alpha_{\text{base}}$ , the signature only contains  $\alpha_{\text{mid}}$ . Then, using  $\alpha_{\text{eval}} := P_\alpha(\phi(i^*))$  and  $\alpha_{\text{mid}}$ , the verifier will be able to recompute  $\alpha_{\text{base}}$ .

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

**Secret key:** Two matrices  $\mathbf{S} \in \mathbb{F}_q^{m \times r}$  and  $\mathbf{C}' \in \mathbb{F}_q^{r \times r(n-r)}$  satisfying  $\mathbf{H} \text{vec}(\mathbf{S} \cdot [\mathbf{I}_r \parallel \mathbf{C}']) = \mathbf{y}$ .

**Step 0: Initialization**

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

**Step 1: Build & Commit to Witness Polynomials**

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

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

6. Sample  $\mathbf{\Gamma} \xleftarrow{\$} \text{PRG}(h_{\text{sh}})$  where  $\mathbf{\Gamma} \in \mathbb{F}_{q^\mu}^{\rho \times mn-k}$ .
7. For each iteration  $e \in [1, \dots, \tau]$ :
  - (a) Compute  $P_E^{(e)}(X)$  by computing
 
$$\mathbf{E}_{\text{mid}}^{(e)} := [\mathbf{S}_{\text{base}}^{(e)} \parallel \mathbf{S}_{\text{base}}^{(e)} \cdot \mathbf{C}'_{\text{base}}^{(e)} + \mathbf{S}^{(e)} \cdot \mathbf{C}'_{\text{base}}^{(e)}],$$

$$\mathbf{E}_{\text{base}}^{(e)} := [\mathbf{0}_{m \times r} \parallel \mathbf{S}_{\text{base}}^{(e)} \cdot \mathbf{C}'_{\text{base}}^{(e)}].$$
  - (b) Compute the polynomial  $P_\alpha^{(e)} := \alpha_{\text{mid}}^{(e)} \cdot X + \alpha_{\text{base}}^{(e)}$  by computing
 
$$\alpha_{\text{mid}}^{(e)} := \mathbf{\Gamma} \cdot (\mathbf{I}_{m \cdot n - k} \parallel \mathbf{H}') \cdot \text{vec}(\mathbf{E}_{\text{mid}}^{(e)}) + \mathbf{v}^{(e)},$$

$$\alpha_{\text{base}}^{(e)} := \mathbf{\Gamma} \cdot (\mathbf{I}_{m \cdot n - k} \parallel \mathbf{H}') \cdot \text{vec}(\mathbf{E}_{\text{base}}^{(e)}) + \mathbf{v}_{\text{base}}^{(e)}.$$
8. Compute  $h_{\text{piop}} = \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, (\alpha_{\text{mid}}^{(e)}, \alpha_{\text{base}}^{(e)})_{e \in [1, \dots, \tau]}).$

**Step 3: Open Random Evaluations**

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

**Algorithm 1.** High level description of Mirath Sign algorithm

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

**Step 0: Initialization**

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

**Step 1: Computing Opened Evaluations**

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

$$\begin{aligned} \mathbf{S}_{\text{eval}}^{(e)} &= \phi(i^{*(e)}) \cdot \mathbf{S}_{\text{aux}}^{(e)} + \sum_{i=1, i \neq i^{*(e)}}^N (\phi(i^{*(e)}) - \phi(i)) \cdot \mathbf{S}_{\text{rnd}, i}^{(e)} \\ \mathbf{C}'_{\text{eval}}^{(e)} &= \phi(i^{*(e)}) \cdot \mathbf{C}'_{\text{aux}}^{(e)} + \sum_{i=1, i \neq i^{*(e)}}^N (\phi(i^{*(e)}) - \phi(i)) \cdot \mathbf{C}'_{\text{rnd}, i}^{(e)} \\ \mathbf{v}_{\text{eval}}^{(e)} &= \sum_{i=1, i \neq i^{*(e)}}^N (\phi(i^{*(e)}) - \phi(i)) \cdot \mathbf{v}_{\text{rnd}, i}^{(e)}. \end{aligned}$$

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

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

7. Sample  $\boldsymbol{\Gamma} \xleftarrow{\$} \text{PRG}(h_{\text{sh}})$  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  $\mathbf{E}_{\text{eval}}^{(e)} := P_E^{(e)}(\phi(i^{*(e)}))$  as
 
$$\mathbf{E}_{\text{eval}}^{(e)} = [\mathbf{S}_{\text{eval}}^{(e)} \cdot \phi(i^{*(e)}) \parallel \mathbf{S}_{\text{eval}}^{(e)} \cdot \mathbf{C}'_{\text{eval}}^{(e)}].$$
  - (b) Compute the evaluation  $\boldsymbol{\alpha}_{\text{eval}}^{(e)} := P_\alpha^{(e)}(\phi(i^{*(e)}))$  as
 
$$\boldsymbol{\alpha}_{\text{eval}}^{(e)} = \mathbf{v}_{\text{eval}} + \boldsymbol{\Gamma} \cdot \left( (\mathbf{I}_{m \cdot n - k} \parallel \mathbf{H}') \cdot \text{vec}(\mathbf{E}_{\text{eval}}^{(e)}) - \mathbf{y} \cdot \phi(i^{*(e)})^2 \right).$$
  - (c) Deduce  $\boldsymbol{\alpha}_{\text{base}}^{(e)}$  as
 
$$\boldsymbol{\alpha}_{\text{base}}^{(e)} = \boldsymbol{\alpha}_{\text{eval}}^{(e)} - \boldsymbol{\alpha}_{\text{mid}}^{(e)} \cdot \phi(i^{*(e)}).$$
9. Compute  $h'_{\text{piop}} = \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, (\boldsymbol{\alpha}_{\text{mid}}^{(e)}, \boldsymbol{\alpha}_{\text{base}}^{(e)})_{e \in [1, \dots, \tau]}).$

**Step 3: Verification**

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

**Algorithm 2.** High level description of Mirath Verify algorithm

## 4 Detailed Description of Mirath

### 4.1 Notations

We let  $y \leftarrow A(x)$  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  $\mathbf{a}[i]$  denote the  $i$ th element of the array  $\mathbf{a}$ . The main variable names employed in the algorithms are collected in Table 2.

Vectors and Matrices:		
$\mathbf{E}$	$\mathbb{F}_q^{m \times n}$	Matrix solution of the Dual Support Modeling.
$\mathbf{S}, \mathbf{C}'$	$\mathbb{F}_q^{m \times r}, \mathbb{F}_q^{r \times (n-r)}$	Matrices of the secret key: $\mathbf{E} = \mathbf{S}[\mathbf{I}_r \ \mathbf{C}']$ .
$\mathbf{H}'$	$\mathbb{F}_q^{(m \cdot n - k) \times k}$	Matrix of the public key.
$\mathbf{y}$	$\mathbb{F}_q^{(mn-k) \times k}$	Vector of the public key: $[\mathbf{I}_{mn-k} \ \mathbf{H}'] \text{vec}(\mathbf{E}) = \mathbf{y}$ .
$\mathbf{v}$	$\mathbb{F}_{q^\mu}^{\rho \times 1}$	Random masking vector.
$\mathbf{\Gamma}$	$\mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$	Challenge matrix.
$\phi(i^*)$	$\mathbb{F}_{q^\mu}$	Evaluation points from the second challenge $\{i^*[e]\}_{e < \tau}$ .
Seeds and Salt:		
seed <sub>pk</sub>	$\{0, 1\}^\lambda$	Seed for the generation of the matrix $\mathbf{H}'$ of the public key.
seed <sub>sk</sub>	$\{0, 1\}^\lambda$	Seed for the generation of the matrices $\mathbf{S}, \mathbf{C}'$ 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 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 <sub>i*</sub>	$\{0, 1\}^{2\lambda\tau}$	Commitments of the unopened BAVC leaves in tree.
$\pi_{\text{BAVC}}$	$\{0, 1\}^{\lambda(T_{\text{open}} + 2\tau)}$	path    commit <sub>i*</sub> .
$h_{\text{com}}$	$\{0, 1\}^{2\lambda}$	Seed commitment (hash digest of commit).
$h_{\text{sh}}$	$\{0, 1\}^{2\lambda}$	Commitment of the witness polynomials.
$h_{\text{piop}}$	$\{0, 1\}^{2\lambda}$	Polynomial proof commitment.
ctr	$\{0, 1\}^{64}$	Counter for grinding and proof-of-work optimizations.
$\sigma$	$\{0, 1\}^*$	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  $a_{\mu-1}x^{\mu-1} + \dots + a_1x + a_0$ , where  $a_0, \dots, a_{\mu-1} \in \mathbb{F}_q$ . We store an element  $a_{\mu-1}x^{\mu-1} + \dots + a_1x + a_0 \in \mathbb{F}_{q^\mu}$  as the integer with binary representation

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

where  $\text{bin}(a_i)$  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.

$q$	$\mu$	$f(x)$
2	8	$x^8 + x^4 + x^3 + x + 1$
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  $\mathbb{M}$  of length  $\text{nb}_c \cdot n_c$ , where  $\text{nb}_c = \lceil n_r \log q / 8 \rceil$ . This array is represented as  $\mathbb{M} = \mathbb{M}_0 \parallel \mathbb{M}_1 \parallel \dots \parallel \mathbb{M}_{n_c-1}$ , where each  $\mathbb{M}_j$  is a byte-array of length  $\text{nb}_c$  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  $0 \leq i \leq \text{nb}_c - 2$ . The  $i$ -th byte  $M_j[i]$  in the byte array  $M_j$  stores the  $p$  entries  $a_{p \cdot i}, a_{p \cdot i + 1}, \dots, a_{p \cdot i + p - 1}$  by defining

$$M_j[i] = \text{bin}(a_{p \cdot i + p - 1}) \parallel \text{bin}(a_{p \cdot i + p - 2}) \parallel \dots \parallel \text{bin}(a_{p \cdot i + 1}) \parallel \text{bin}(a_{p \cdot i}),$$

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

$$M_j[\text{nb}_c - 1] = \mathbf{0} \parallel \text{bin}(a_{p(\text{nb}_c - 1) + p' - 1}) \parallel \dots \parallel \text{bin}(a_{p(\text{nb}_c - 1) + 1}) \parallel \text{bin}(a_{p(\text{nb}_c - 1)}),$$

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  $A$  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(A,  $\mathbb{F}^{n_r \times n_c}$ )
1 :  $p \leftarrow 8 / \log q$   $\triangleright$  Maximun number of elements stored in one byte.
2 :  $\text{mask} \leftarrow 255$ 
3 : if  $\mathbb{F} = \mathbb{F}_q$  then
4 :    $\text{nb}_c \leftarrow \lceil n_r / p \rceil$   $\triangleright$  Number of bytes used to store one column.
5 :    $a \leftarrow n_r \bmod p$ 
6 :   if  $a > 0$  then
7 :      $\text{mask} \leftarrow \lfloor \text{mask} / 2^{(p-a) \cdot \log q} \rfloor$ 
8 :   for  $j$  from 1 to  $n_c$  do
9 :      $A[j \cdot \text{nb}_c - 1] \leftarrow A[j \cdot \text{nb}_c - 1] \otimes \text{mask}$   $\triangleright \otimes$  means bitwise multiplication
10 : if  $\mathbb{F} = \mathbb{F}_{q^\mu}$  then
11 :    $\text{nb}_e \leftarrow \lceil \mu / p \rceil$   $\triangleright$  Number of bytes used to store one  $\mathbb{F}_{q^\mu}$  element.
12 :    $a \leftarrow \mu \bmod p$ 
13 :   if  $a > 0$  then
14 :      $\text{mask} \leftarrow \lfloor \text{mask} / 2^{(p-a) \cdot \log q} \rfloor$ 
15 :   for  $j$  from 0 to  $n_c n_r - 1$  do
16 :      $A[j \cdot \text{nb}_e] \leftarrow A[j \cdot \text{nb}_e] \otimes \text{mask}$   $\triangleright \otimes$  means bitwise multiplication
17 : return 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  $M$  of length  $\text{nb}_c \cdot n_c$ , where  $\text{nb}_c = \lceil \mu \log q / 8 \rceil n_r$ . The  $(i, j)$  entry of  $M$  is stored in the  $s = \lceil \mu \log q / 8 \rceil$  bytes  $M[\text{nb}_c \cdot j + s \cdot i], \dots, M[\text{nb}_c \cdot j + s \cdot i + s - 1]$  as explained in Section 4.2.

#### 4.4 Hash Functions and Commitments

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

$$\begin{aligned}\text{Hash}_1(\text{data}) &:= \text{Hash}(1, \text{data}) \\ \text{Hash}_2(\text{data}) &:= \text{Hash}(2, \text{data}) \\ \text{Hash}_3(\text{data}) &:= \text{Hash}(3, \text{data})\end{aligned}$$

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

$\text{Hash}_1$  is used in *CommitWitnessPolynomials* and *ComputeEvaluations*.  $\text{Hash}_2$  is used in *Sign* and *Verify*.  $\text{Hash}_3$  is used in *BAVC.Commit* and *BAVC.Reconstruct*.

**Commit Function.** In *BAVC.Commit* and *BAVC.Reconstruct* we compute a commitment for every seed  $\text{seed}_i$  by using function *Commit* defined as

$$\text{Commit}(\text{salt}, \text{seed}, \text{idx}) := \begin{cases} \text{ExpandSeed}_3(\text{salt}, \text{seed}, \text{idx}) & \text{for Rijndael commits} \\ \text{Hash}_3(\text{salt} \parallel \text{bin}_{32}(\text{idx}) \parallel \text{seed}) & \text{for SHA3 commits.} \end{cases}$$

In the case of Rinjdael commits the output of  $\text{ExpandSeed}_3$  is parsed as one binary string of size  $2\lambda$ .

#### 4.5 Randomness Generation and Sampling

**Random Objects.**  $\text{TRG.Seed}(\lambda)$  returns a random binary string of length  $\lambda$ . 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- $\lambda$*  or the *XOF Shake*.

$\text{Encrypt-}\lambda(\text{key}, \text{msg})$  output the encryption  $c \in \{0, 1\}^\lambda$  of the block  $\text{msg} \in \{0, 1\}^\lambda$  using the key  $\text{key} \in \{0, 1\}^\lambda$ . The selection of the particular block cipher depends on the security level. In *Mirath*  $\text{Encrypt-128} = \text{AES-128}$  and  $\text{Encrypt-}\lambda = \text{Rijndael-}\lambda$  for  $\lambda = 192, 256$ . This functionality is used in Algorithm 2 and Algorithm 3.



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 2** Children nodes generation.

---

<p><code>ExpandSeed<sub>a</sub>(salt, seed, idx)</code></p> <hr style="border: 0; border-top: 1px solid black; margin: 5px 0;"/> <pre style="margin: 0;"> 1 : salt<sub>0</sub> ← salt[0, λ]  ▷ First λ bits in salt. 2 : msg<sub>0</sub> ← salt<sub>0</sub> ⊕ (bin<sub>λ-40</sub>(a)    bin<sub>32</sub>(idx)    bin<sub>8</sub>(0)) 3 : msg<sub>1</sub> ← salt<sub>0</sub> ⊕ (bin<sub>λ-40</sub>(a)    bin<sub>32</sub>(idx)    bin<sub>8</sub>(1)) 4 :      ▷ bin<sub>x</sub>(i) is the x-bit representation of i. 5 : seed<sub>l</sub> ← Encrypt-λ(seed, msg<sub>0</sub>) 6 : seed<sub>r</sub> ← Encrypt-λ(seed, msg<sub>1</sub>) 7 : return (seed<sub>l</sub>, seed<sub>r</sub>) </pre>
---

---

`ExpandSeeda(salt, seed, idx)` returns a pseudorandom element in  $\{0, 1\}^\lambda \times \{0, 1\}^\lambda$  built from a salt `salt`  $\in \{0, 1\}^{128}$ , a seed `seed`  $\in \{0, 1\}^\lambda$  and a nonnegative integer `idx`  $< N$ . It is described in Algorithm 2, and used in `BAVC.Commit` and `BAVC.Reconstruct`.

**Algorithm 3** Pseudorandom element in  $\mathbb{F}_q^{m \times r} \times \mathbb{F}_q^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$ .

---

ExpandSeedShares(salt, seed)

---

```

1 : salt0 ← salt[0, λ] ▷ First λ bits in salt.
2 : p ← 8 / log q
3 : nbs ← ⌈m/p⌉r ▷ Number of bytes to store Srnd ∈  $\mathbb{F}_q^{m \times r}$ .
4 : nbc ← ⌈r/p⌉ · (n - r) ▷ Number of bytes to store Crnd ∈  $\mathbb{F}_q^{r \times (n-r)}$ .
5 : nbv ← ⌈μ/p⌉ρ ▷ Number of bytes to store vrnd ∈  $\mathbb{F}_{q^\mu}^{\rho \times 1}$ .
6 : nb ← nbs + nbc + nbv
7 : nenc ← ⌈nb · 8/λ⌉
8 : A ← ∅
9 : for i from 0 to nenc - 1 do
10 :   A ← A || Encrypt-λ(seed, salt0 ⊕ binλ(i))
11 :   ▷ binλ(i) is the λ-bit representation of i.
12 :   Srnd ← SetToMatrix(A[0], ..., A[nbs - 1],  $\mathbb{F}_q^{m \times r}$ )
13 :   Crnd ← SetToMatrix(A[nbs], ..., A[nbs + nbc - 1],  $\mathbb{F}_q^{r \times (n-r)}$ )
14 :   vrnd ← SetToMatrix(A[nbs + nbc] || ... || A[nb - 1],  $\mathbb{F}_{q^\mu}^{\rho \times 1}$ )
15 : return (Srnd, Crnd, vrnd)

```

---

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  $\text{salt} \in \{0, 1\}^{128}$  and a seed  $\text{seed} \in \{0, 1\}^\lambda$ . Algorithm 3 describes this routine, which is used during signing and verification within **CommitWitnessPolynomials** and **ComputeEvaluations**, respectively.

**Algorithm 4** Routine ExpandSeedPublicMatrix.

---

ExpandSeedPublicMatrix(seed<sub>pk</sub>)

---

```

1 : p ← 8 / log q
2 : nb ← ⌈(mn - k)/p⌉k ▷ Number of bytes to store H' ∈  $\mathbb{F}_q^{(mn-k) \times k}$ .
3 : prg ← Shake.Init()
4 : prg ← Shake.Absorb(prg, seedpk)
5 : H ← Shake.Squeeze(prg, nb)
6 : H' ← SetToMatrix(H,  $\mathbb{F}_q^{(mn-k) \times k}$ )
7 : return H'

```

---

ExpandSeedPublicMatrix(seed<sub>pk</sub>) returns the matrix **H**' ∈  $\mathbb{F}_q^{(mn-k) \times k}$  which is expended from a seed seed<sub>pk</sub>. It is described in Algorithm 4, and it is used in

**KeyGen** and **Verify** within the **DecompressPK** function.

---

**Algorithm 5** Routine **ExpandSeedSecretMatrices**.

---

```

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

```

---

**ExpandSeedSecretMatrices**(seed) returns the secret matrices  $\mathbf{S} \in \mathbb{F}_q^{m \times r}$  and  $\mathbf{C}' \in \mathbb{F}_q^{r \times (n-r)}$ . It is used in **KeyGen** and **Sign** within the **DecompressSK** function that also returns the matrices  $\mathbf{H}' \in \mathbb{F}_q^{(mn-k) \times k}$  and the public key **pk**.

---

**Algorithm 6** Expand challenge evaluation points.

---

```

ExpandChallengeEvaluationPoints( $h$ , ctr)
1 :  $\text{nbits} \leftarrow (\tau \cdot \log N) + w$ 
2 :  $\text{nb} \leftarrow \lceil \text{nbits}/8 \rceil$ 
3 :  $\text{prg} \leftarrow \text{Shake.Init}()$ 
4 :  $\text{prg} \leftarrow \text{Shake.Absorb}(\text{prg}, h \parallel \text{bin}_{64}(\text{ctr}))$ 
5 :  $\text{rnd} \leftarrow \text{Shake.Squeeze}(\text{prg}, \text{nb})$ 
6 : for  $e$  from 0 to  $(\tau - 1)$  do
7 :    $i^*[e] \leftarrow \text{rnd}[e \cdot \log N, (e + 1) \cdot \log N - 1] \triangleright \log N$  bits of  $\text{rnd}$  assigned to  $i^*[e]$ .
8 :  $v_{\text{grinding}} \leftarrow \text{rnd}[\tau \cdot \log N, \tau \cdot \log N + w - 1]$ 
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 7** Challenge matrix  $\mathbf{I} \in \mathbb{F}_{q^\mu}^{(mn-k) \times k}$ .

---

ExpandChallengeMatrix( $h$ )

---

```

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

```

---

ExpandChallengeMatrix( $h$ ) returns the challenge matrix  $\mathbf{I} \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$ , 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 8** Routine BAVC.Commit.

---

BAVC.Commit(salt, rseed)

---

```

1 : tree[0]  $\leftarrow \text{rseed}$ 
2 : for  $i$  from 0 to  $(\tau \cdot N - 2)$  do
3 :   (tree[2i + 1], tree[2i + 2])  $\leftarrow \text{ExpandSeed}_4(\text{salt}, \text{tree}[i], i)$ 
4 : for  $e$  from 0 to  $(\tau - 1)$  do
5 :   for  $i$  from 0 to  $(N - 1)$  do
6 :     seeds[ $e$ ][ $i$ ]  $\leftarrow \text{tree}[(\tau \cdot N - 1) + (i \cdot \tau + e)]$ 
7 :     commit[ $e$ ][ $i$ ]  $\leftarrow \text{Commit}(\text{salt}, \text{seeds}[e][i], (\tau \cdot N - 1) + i \cdot \tau + e)$ 
8 :    $h_{\text{com}} \leftarrow \text{Hash}_3(\{\text{commit}[e][i]\}_{e,i})$ 
9 :   key  $\leftarrow \text{tree} \parallel \text{commit}$ 
10 : return (seeds,  $h_{\text{com}}$ , key)

```

---

**Algorithm 9** Routine BAVC.Open.

---

```

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

```

---

Algorithm 8 describes BAVC.Commit, i.e. the Commit method of the BAVC scheme used in Mirath. This function outputs  $(\text{seeds}, h_{\text{com}}, \text{key})$ , where  $\text{seeds} = \{\text{seeds}[e][i]\}_{e < \tau, i < N}$  is a set of  $\tau \cdot N$  seeds in  $\{0, 1\}^\lambda$ ,  $h_{\text{com}}$  is a commitment to  $\text{seeds}$ , and  $\text{key}$  contains a binary tree  $\text{tree}$  with leaves given by  $\text{seeds}$  and a set  $\text{commit}$  of  $\tau \cdot N$  elements in  $\{0, 1\}^{2\lambda}$  containing one commit per seed in  $\text{seeds}$ . BAVC.Commit is used during signing within the CommitWitnessPolynomials algorithm.

Algorithm 9 describes BAVC.Open, i.e. the Open method of the BAVC scheme used in Mirath, which outputs the opening  $\pi_{\text{BAVC}}$  for the leaves in  $\text{tree}$  with indexes in  $\{i^*[e]\}_{e < \tau}$ . BAVC.Open is used in OpenRandomEvaluations.

Algorithm 10 describes BAVC.Reconstruct, i.e. the Reconstruct method of the BAVC scheme used in Mirath. This function outputs  $(h_{\text{com}}, \text{seeds})$ , where  $h_{\text{com}}$  is the reconstruction of a commitment made for a set of  $N$  seeds, and  $\text{seeds}$  is the set of  $N - \tau$  opened seeds  $\{\text{seeds}[e][i]\}_{0 \leq e < \tau, i \neq i^*[e]}$ .

#### 4.7 Commitment to Polynomials

This section details how to commit to the witness polynomials  $P_S(X) = \mathbf{S} \cdot X + \mathbf{S}_{\text{base}}$ ,  $P_{C'}(X) = \mathbf{C}' \cdot X + \mathbf{C}'_{\text{base}}$  and the masking polynomial  $P_v(X) = \mathbf{v} \cdot X + \mathbf{v}_{\text{base}}$  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 10** Routine BAVC.Reconstruct.

---

```

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

1 : (path,  $\text{commit}_{i^*}$ )  $\leftarrow \pi_{\text{BAVC}}$ 
2 : hidden  $\leftarrow \{(\tau \cdot N - 1) + (i^*[e] \cdot \tau + e) : e \in \{0, \dots, \tau - 1\}\}$ 
3 : revealed  $\leftarrow \{\tau \cdot N - 1, \dots, 2 \cdot \tau \cdot N - 2\} \setminus \text{hidden}$ 
4 : for  $i$  from  $(\tau \cdot N - 2)$  downto 0 do
5 :   if  $(2i + 1) \in \text{revealed}$  and  $(2i + 2) \in \text{revealed}$  then
6 :     revealed  $\leftarrow (\text{revealed} \setminus \{2i + 1, 2i + 2\}) \cup \{i\}$ 
7 :   tree[0], ..., tree[ $2 \cdot \tau \cdot N - 2$ ]  $\leftarrow \emptyset, \dots, \emptyset$ 
8 :   for  $i$  from 0 to  $(2 \cdot \tau \cdot N - 2)$  do
9 :     if  $i \in \text{revealed}$  then
10 :      (tree[ $i$ ], path)  $\leftarrow \text{path}$ 
11 :     if tree[ $i$ ]  $\neq \emptyset$  and  $i < \tau \cdot N - 1$  then
12 :      (nodes[ $2i + 1$ ], nodes[ $2i + 2$ ])  $\leftarrow \text{ExpandSeed}_4(\text{salt}, \text{nodes}[i], i)$ 
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 :         seeds[ $e$ ][ $i$ ]  $\leftarrow \text{nodes}[(\tau \cdot N - 1) + (i \cdot \tau + e)]$ 
17 :         com[ $e$ ][ $i$ ]  $\leftarrow \text{Commit}(\text{salt}, \text{seeds}[e][i], (\tau \cdot N - 1) + i \cdot \tau + e)$ 
18 :       else
19 :         seeds[ $e$ ][ $i$ ]  $\leftarrow \emptyset$ 
20 :         (com[ $e$ ][ $i$ ] ||  $\text{commit}_{i^*}$ )  $\leftarrow \text{commit}_{i^*}$ 
21 :    $h_{\text{com}} \leftarrow \text{Hash}_3(\{\text{com}[e][i]\}_{e,i})$ 
22 :   return ( $h_{\text{com}}$ , seeds)

```

---

**Algorithm 12** Open random evaluations.

---

```

OpenRandomEvaluations(key,  $h_{\text{piop}}$ )

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

```

---

**Algorithm 11** Commit to witness  $(P_S, P_{C'})$  and masking  $P_v$  polynomials.

---

CommitWitnessPolynomials(salt, rseed,  $S, C'$ )

---

```

1 : (seeds,  $h_{com}$ , key)  $\leftarrow$  BAVC.Commit(salt, rseed)
2 : for  $e$  from 0 to  $(\tau - 1)$  do
3 :   ( $S_{acc}, C'_{acc}, v$ )  $\leftarrow$  (0, 0, 0)  $\triangleright S_{acc} \in \mathbb{F}_q^{m \times r}, C'_{acc} \in \mathbb{F}_q^{r \times (n-r)}, v \in \mathbb{F}_{q^\mu}^{\rho \times 1}$ 
4 :   ( $S_{base}, C'_{base}, v_{base}$ )  $\leftarrow$  (0, 0, 0)  $\triangleright S_{base} \in \mathbb{F}_{q^\mu}^{m \times r}, C'_{base} \in \mathbb{F}_{q^\mu}^{r \times (n-r)}, v_{base} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$ 
5 :   for  $i$  from 0 to  $(N - 1)$  do
6 :     ( $S_{rnd}, C'_{rnd}, v_{rnd}$ )  $\leftarrow$  ExpandSeedShares(salt, seeds[ $e$ ][ $i$ ])
7 :     ( $S_{acc}, C'_{acc}, v_{acc}$ )  $+=$  ( $S_{rnd}, C'_{rnd}, v_{rnd}$ )
8 :     ( $S_{base}, C'_{base}, v_{base}$ )  $-=$  ( $\phi(i) \cdot S_{rnd}, \phi(i) \cdot C'_{rnd}, \phi(i) \cdot v_{rnd}$ )
            $\triangleright \phi : \{0, \dots, q^\mu - 1\} \rightarrow \mathbb{F}_{q^\mu}$ 
9 :   aux[ $e$ ]  $\leftarrow$  ( $S - S_{acc}, C' - C'_{acc}$ )
10 :  base[ $e$ ]  $\leftarrow$  ( $S_{base}, C'_{base}, v_{base}$ )
11 :  v[ $e$ ]  $\leftarrow$  vacc
12 :   $h_{sh} \leftarrow$  Hash1(salt,  $h_{com}$ , aux[0],  $\dots$ , aux[ $\tau - 1$ ])
13 : return (base, v,  $h_{sh}$ , key, aux)

```

---

Algorithm 11 describes the routine **CommitWitnessPolynomials** that commits to  $\tau$  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 12 details the routine **OpenRandomEvaluations** used in **Sign** to output  $(ctr, \pi_{BAVC})$ , which contains the necessary information to compute the evaluations  $S_{eval} = P_S(\text{point}[e])$ ,  $C'_{eval} = P_{C'}(\text{point}[e])$ , and  $v_{eval} = P_v(\text{point}[e])$  for any  $e < \tau$  and to check the consistency of the commitments made at Step 1 of the Mirath protocol. The value  $\pi_{BAVC}$  is the output of the subroutine **BAVC.Open**, while  $ctr$  incorporates the output from the grinding and proof-of-work optimizations.

Algorithm 13 details the routine **ComputeEvaluations** used in **Verify** to compute the evaluations  $S_{eval} = P_S(\text{point}[e])$ ,  $C'_{eval} = P_{C'}(\text{point}[e])$  and  $v_{eval} = P_v(\text{point}[e])$  for any  $e < \tau$ , and to check the consistency of the commitment made at Step 1 of the Mirath protocol.

**Algorithm 13** Compute evaluations of  $P_S$ ,  $P_{C'}$  and  $P_v$  on the opened points.

---

```

ComputeEvaluations(salt, ctr,  $h_{\text{piop}}$ ,  $\pi_{\text{BAVC}}$ , aux)
1 :  $(\{i^*[e]\}_{e < \tau}, v_{\text{grinding}}) \leftarrow \text{ExpandChallengeEvaluationPoints}(h_{\text{piop}}, \text{ctr})$ 
2 :  $h_{\text{com}}, \text{seeds} \leftarrow \text{BAVC.Reconstruct}(\{i^*[e]\}_{e < \tau}, \pi_{\text{BAVC}}, \text{salt})$ 
3 :  $h_{\text{sh}} \leftarrow \text{Hash}_1(\text{salt}, h_{\text{com}}, \text{aux}[0], \dots, \text{aux}[\tau - 1])$ 
4 : for  $e$  from 0 to  $(\tau - 1)$  do
5 :    $(S_{\text{eval}}[e], C'_{\text{eval}}[e], v_{\text{eval}}[e]) = (0, 0, 0)$ 
6 :    $\triangleright (S_{\text{eval}}[e], C'_{\text{eval}}[e], v_{\text{eval}}[e]) \in \mathbb{F}_{q^\mu}^{m \times r} \times \mathbb{F}_{q^\mu}^{r \times (n-r)} \times \mathbb{F}_{q^\mu}^{\rho \times 1}$ 
7 :   for  $i$  from 0 to  $N - 1$  do
8 :     if  $i \neq i^*[e]$  then
9 :        $(S_{\text{rnd}}, C'_{\text{rnd}}, v_{\text{rnd}}) \leftarrow \text{ExpandSeedShares}(\text{salt}, \text{seeds}[e][i])$ 
10 :       $a \leftarrow \phi(i^*[e]) - \phi(i) \triangleright \phi : \{0, \dots, q^\mu - 1\} \rightarrow \mathbb{F}_{q^\mu}$ 
11 :       $(S_{\text{eval}}[e], C'_{\text{eval}}[e], v_{\text{eval}}[e]) += (aS_{\text{rnd}}, aC'_{\text{rnd}}, av_{\text{rnd}})$ 
12 :    $\text{point}[e] \leftarrow \phi(i^*[e])$ 
13 :    $(S_{\text{aux}}, C'_{\text{aux}}) \leftarrow \text{aux}[e]$ 
14 :    $(S_{\text{eval}}[e], C'_{\text{eval}}[e]) += (\text{point}[e] \cdot S_{\text{aux}}, \text{point}[e] \cdot C'_{\text{aux}})$ 
15 :    $\text{eval}[e] \leftarrow (S_{\text{eval}}[e], C'_{\text{eval}}[e], v_{\text{eval}}[e])$ 
16 : return  $(\text{eval}, \text{point}, h_{\text{sh}}, v_{\text{grinding}})$ 

```

---

#### 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 14 describes the routine **ComputePolynomialProof** used in **Sign** to compute the  $\tau$  polynomial proof  $P_\alpha(X) = \alpha_{\text{mid}} \cdot X + \alpha_{\text{base}}$  from the  $\tau$  committed polynomials  $P_S(X) = S \cdot X + S_{\text{base}}$ ,  $P_C(X) = C' \cdot X + C'_{\text{base}}$ ,  $P_v(X) = v \cdot X + v_{\text{base}}$  and the matrices  $\mathbf{I} \in \mathbb{F}_{q^\mu}^{\rho \times (m \cdot n - k)}$  and  $\mathbf{H}' \in \mathbb{F}_q^{(mn-k) \times k}$ .

Algorithm 15 describes **RecomputePolynomialProof**, used in **Verify** to recover the polynomial proofs  $P_\alpha(X)$  from the different evaluations  $S_{\text{eval}} = P_S(\text{point}[e])$ ,  $C'_{\text{eval}} = P_C(\text{point}[e])$  and  $v_{\text{eval}} = P_v(\text{point}[e])$  with a given  $\text{point}[0], \dots, \text{point}[\tau - 1] \in S \subset \mathbb{F}_{q^\mu}$ .



---

**Algorithm 14** Computation of the polynomial  $P_\alpha$ .

---

ComputePolynomialProof(base,  $v$ ,  $S$ ,  $C'$ ,  $\Gamma$ ,  $H'$ )

---

```

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

```

---



---

**Algorithm 15** Reconstruction of the polynomial  $P_\alpha$ .

---

RecomputePolynomialProof(point, eval,  $\Gamma$ , ( $H'$ ,  $y$ ),  $\alpha_{\text{mid}}$ )

---

```

1 :  $S_{\text{eval}}, C'_{\text{eval}}, v_{\text{eval}} \leftarrow \text{eval}$ 
2 :  $E_{\text{eval}} \leftarrow (\text{point} \cdot S_{\text{eval}} \mid S_{\text{eval}} C'_{\text{eval}}) \triangleright E_{\text{eval}} \in \mathbb{F}_{q^\mu}^{m \times n}$ 
3 :  $e \leftarrow \text{vec}(E_{\text{eval}}) \triangleright e \in \mathbb{F}_{q^\mu}^{mn}$  with the entries of  $E_{\text{eval}}$  in a column-major order
4 :  $(e_A, e_B) \leftarrow \text{Split}(e) \triangleright e_A \in \mathbb{F}_{q^\mu}^{(mn-k) \times 1}, e_B \in \mathbb{F}_{q^\mu}^{k \times 1}$ 
5 :  $\alpha_{\text{eval}} \leftarrow \Gamma(e_A + H' e_B - y \cdot \text{point}^2) + v_{\text{eval}} \triangleright \alpha_{\text{eval}} \in \mathbb{F}_{q^\mu}^{\rho \times 1}$ 
6 :  $\alpha_{\text{base}} \leftarrow \alpha_{\text{eval}} - \alpha_{\text{mid}} \cdot \text{point}$ 
7 : return  $\alpha_{\text{base}}$ 

```

---

#### 4.9 Parsing of the Signature

In Algorithm 16 and Algorithm 17 we give the details of packing/unpacking the signature in Mirath. This packing is performed tightly in order to efficiently avoid 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 16** Unparse signature.

---

```

UnparseSignature (salt, ctr,  $h_{\text{piop}}$ ,  $\pi_{\text{BAVC}}$ , aux,  $\alpha_{\text{mid}}$ )
1 :  $\sigma \leftarrow \text{salt} \parallel \text{bin}_{64}(\text{ctr}) \parallel h_{\text{piop}} \parallel \pi_{\text{BAVC}}$ 
    $\triangleright \text{bin}_{64}(\text{ctr})$  is the 64-bit representation of ctr.
2 :  $\sigma \leftarrow \sigma \parallel \text{ToBitString}(\alpha_{\text{mid}}[0], \dots, \alpha_{\text{mid}}[\tau - 1])$   $\triangleright \alpha_{\text{mid}}[e] \in \mathbb{F}_q^{\rho \times 1}$ 
3 : for  $e$  from 0 to  $(\tau - 1)$  do
4 :    $(S_{\text{aux}}[e] \parallel C'_{\text{aux}}[e]) \leftarrow \text{aux}[e]$   $\triangleright S_{\text{aux}}[e] \in \mathbb{F}_q^{m \times r}, C'_{\text{aux}}[e] \in \mathbb{F}_q^{r \times (n-r)}$ 
5 :  $\sigma \leftarrow \sigma \parallel \text{ToBitString}(S_{\text{aux}}[0], \dots, S_{\text{aux}}[\tau - 1])$ 
6 :  $\sigma \leftarrow \sigma \parallel \text{ToBitString}(C'_{\text{aux}}[0], \dots, C'_{\text{aux}}[\tau - 1])$ 
7 : return  $\sigma$ 

```

---

**Algorithm 17** Parse signature.

---

```

ParseSignature( $\sigma$ )
1 : salt  $\leftarrow \sigma[0, 2\lambda - 1]$ 
2 : ctr  $\leftarrow \sigma[\lambda, \lambda + 63]$ 
3 :  $h_{\text{piop}} \leftarrow \sigma[\lambda + 64, 3\lambda + 63]$ 
4 :  $\pi_{\text{BAVC}} \leftarrow \sigma[3\lambda + 64, \lambda(3 + T_{\text{open}} + 2\tau) + 63]$ 
5 :  $i_b \leftarrow \lambda(3 + T_{\text{open}} + 2\tau) + 64$ 
6 :  $\triangleright$  s_bit_size, c_bit_size and a_bit_size are the bit-sizes of  $S_{\text{aux}}$ ,  $C'_{\text{aux}}$  and  $\alpha_{\text{mid}}$ , respectively.
7 :  $\alpha_{\text{mid}} \leftarrow \text{AlphaFromBits}(\sigma[i_b, \text{a\_bit\_size}])$   $\triangleright \alpha_{\text{mid}} \in (\mathbb{F}_q^{\rho \times 1})^\tau$ 
8 :  $i_b \leftarrow i_b + \text{a\_bits\_size}$ 
9 :  $S_{\text{aux}} \leftarrow \text{SFromBits}(\sigma[i_b, \text{s\_bits\_size}])$   $\triangleright S_{\text{aux}} \in (\mathbb{F}_q^{m \times r})^\tau$ 
10 :  $i_b \leftarrow i_b + \text{s\_bit\_size}$ 
11 :  $C'_{\text{aux}} \leftarrow \text{CFromBits}(\sigma[i_b, \text{c\_bit\_size}])$   $\triangleright C'_{\text{aux}} \in (\mathbb{F}_q^{r \times (n-r)})^\tau$ 
12 : for  $e$  from 0 to  $(\tau - 1)$  do
13 :   aux[e]  $\leftarrow (S_{\text{aux}}[e] \parallel C'_{\text{aux}}[e])$ 
14 : return (salt, ctr,  $h_{\text{piop}}$ ,  $\pi_{\text{BAVC}}$ , aux,  $\alpha_{\text{mid}}$ )

```

---

**4.10 Key Generation**

The key generation algorithm is described in Algorithm 19. The bit size of the public key is  $|\text{seed}_{\text{pk}}| + |\mathbf{y}| = \lambda + (mn - k) \log q$  and the bit size of the secret key is  $|\text{seed}_{\text{pk}}| + |\text{seed}_{\text{sk}}| = 2\lambda$ .

**Algorithm 18** Routine ComputeY.

---

 ComputeY( $S, C', H'$ )
 

---

```

1 :  $E \leftarrow (S \parallel SC')$   $\triangleright E \in \mathbb{F}_q^{m \times n}$ 
2 :  $e \leftarrow \text{vec}(E)$   $\triangleright e \in \mathbb{F}_q^{mn}$  with the entries of  $E$  in a column-major order
3 :  $(e_A, e_B) \leftarrow \text{Split}(e)$   $\triangleright e_A \in \mathbb{F}_q^{(mn-k) \times 1}, e_B \in \mathbb{F}_q^{k \times 1}$ 
4 :  $y \leftarrow e_A + H' e_B$   $\triangleright y \in \mathbb{F}_q^{(mn-k) \times 1}$ 
5 : return  $y$ 
```

---

**Algorithm 19** Key generation algorithm.

---

 KeyGen()
 

---

```

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

---

The key decompressing routines are shown in Algorithms 20 and 21.

**Algorithm 20** Decompress public key routine.

---

 DecompressPK(pk)
 

---

```

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

---

**Algorithm 21** Decompress secret key routine.

---

 DecompressSK(sk)
 

---

```

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

---

#### 4.11 Signing

Algorithm 22 specifies Mirath's signing algorithm **Sign**. The maximum bit size of a Mirath signature  $\sigma$  is

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

---

**Algorithm 22** Signing algorithm.

---

**Sign**(sk, msg)

*// Step 0: Initialization.*

1 :  $(H', \text{pk}, S, C') \leftarrow \text{DecompressSK}(\text{sk})$

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

3 :  $\text{rseed} \leftarrow \text{TRG.Seed}(\lambda)$

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

4 :  $(\text{base}, v, h_{\text{sh}}, \text{key}, \text{aux}) \leftarrow \text{CommitWitnessPolynomials}(\text{salt}, \text{rseed}, S, C')$

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

5 :  $\Gamma \leftarrow \text{ExpandChallengeMatrix}(h_{\text{sh}}) \triangleright \Gamma \in \mathbb{F}_{q^\mu}^{\rho \times (mn-k)}$

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

7 :    $(\alpha_{\text{mid}}[e], \alpha_{\text{base}}[e]) \leftarrow \text{ComputePolynomialProof}(\text{base}[e], v[e]S, C', \Gamma, H')$

8 :  $h_{\text{piop}} \leftarrow \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}},$   
        $\alpha_{\text{mid}}[0], \alpha_{\text{base}}[0], \dots, \alpha_{\text{mid}}[\tau - 1], \alpha_{\text{base}}[\tau - 1])$

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

9 :  $(\text{ctr}, \pi_{\text{BAVC}}) \leftarrow \text{OpenRandomEvaluations}(\text{key}, h_{\text{piop}})$

10 :  $\sigma \leftarrow \text{UnparseSignature}(\text{salt}, \text{ctr}, h_{\text{piop}}, \pi_{\text{BAVC}}, \text{aux}, \alpha_{\text{mid}})$

11 : **return**  $\sigma$

---

#### 4.12 Verification

Algorithm 23 specifies Mirath's verification algorithm **Verify**.

---

**Algorithm 23** Verification algorithm.

---

```

Verify(pk, σ, msg)
    // Step 0: Initialization (parsing and expansion).
1 : (salt, ctr, hpiop, πBAVC, aux, αmid) ← ParseSignature(σ)
2 : (H', y) ← DecompressPK(pk)

    // Step 1: Computing Opened Evaluations.
3 : (evals, point, hsh, vgrinding) ← ComputeEvaluations(salt, ctr, hpiop, πBAVC, aux)

    // Step 2: Recomputation of the Polynomial Proof Pα(X).
4 : Γ ← ExpandChallengeMatrix(hpiop) ▷ Γ ∈ ℔qμρ × (mn-k)
5 : for e from 0 to (τ - 1) do
6 :   αbase[e] ← RecomputePolynomialProof(point[e], evals[e],
      Γ, (H', y), αmid[e])

    // Step 3: Verification.
7 : h'piop ← Hash2(pk, salt, msg, hsh, αmid[0], αbase[0],
      ..., αmid[τ - 1], αbase[τ - 1])
8 : return (hpiop  $\stackrel{?}{=}$  h'piop) and (vgrinding  $\stackrel{?}{=}$  0)

```

---

## 5 Parameter Sets

We provide several parameter sets using the nomenclature **Mirath-Xy-z** where  $X \in \{1, 3, 5\}$  denotes the security level,  $y \in \{a, b\}$  corresponds to parameters using respectively  $q = 16$  and  $q = 2$  for the base field  $\mathbb{F}_q$  and  $z \in \{\text{short}, \text{fast}\}$  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 <sup>2</sup> [18], which considers all the classical attacks described in Section 8. The bit security estimates given in Tables 4 and 5 are computed by configuring the estimator with all values default but the matrix multiplication constant ( $\omega$  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** ( $q = 2$ )

<sup>2</sup> 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 and Table 7.

Instance	$q$	$\mu$	$\rho$	$\tau$	$N$	$T_{open}$	$w$
Mirath-1a-short	16	3	11	12	$2^{11}$	116	8
Mirath-1a-fast	16	2	16	17	$2^8$	118	9
Mirath-3a-short	16	3	16	19	$2^{11}$	184	2
Mirath-3a-fast	16	2	24	27	$2^8$	178	3
Mirath-5a-short	16	3	22	25	$2^{11}$	240	6
Mirath-5a-fast	16	2	32	36	$2^8$	242	4

**Table 6.** MPC parameters used in **Mirath-a** ( $q = 16$ )

Instance	$q$	$\mu$	$\rho$	$\tau$	$N$	$T_{open}$	$w$
Mirath-1a-short	2	12	11	12	$2^{11}$	116	8
Mirath-1a-fast	2	8	16	17	$2^8$	118	9
Mirath-3a-short	2	12	16	19	$2^{11}$	176	2
Mirath-3a-fast	2	8	24	27	$2^8$	178	3
Mirath-5a-short	2	12	22	25	$2^{11}$	240	6
Mirath-5a-fast	2	8	32	36	$2^8$	242	4

**Table 7.** MPC parameters used in **Mirath-b** ( $q = 2$ )

### 5.3 Key and Signature Sizes

Public key, secret key and signatures size of **Mirath** are given in Tables 8 and 9.

Instance	sk size [B]	pk size [B]	$\sigma$ size [B]
Mirath-1a-short	32	73	3 182
Mirath-1a-fast	32	73	3 728
Mirath-3a-short	48	107	7 456
Mirath-3a-fast	48	107	8 548
Mirath-5a-short	64	147	13 091
Mirath-5a-fast	64	147	15 440

**Table 8.** Keys and signature sizes of **Mirath-a** ( $q = 16$ )

Instance	sk size [B]	pk size [B]	$\sigma$ size [B]
Mirath-1b-short	32	57	2 990
Mirath-1b-fast	32	57	3 456
Mirath-3b-short	48	84	6 825
Mirath-3b-fast	48	84	7 924
Mirath-5b-short	64	112	12 229
Mirath-5b-fast	64	112	14 198

**Table 9.** Keys and signature sizes of **Mirath-b** ( $q = 2$ )



## 6 Performance Analysis

This section provides performance measurements for **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  $i = 25$  random instances. In addition, to minimize biases from background tasks running on the benchmark platform, each instance has been repeated  $i$  times and averaged.

**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/Rijndael or SHAKE. In addition, we provide two variants that differs on how their commitments are instantiated namely either with AES/Rijndael or SHA3.

**Reference Implementation.** The purpose of our reference implementation is to help understand the scheme. As such, this implementation is not representative of the performance that **Mirath** can achieve and does not provide any guarantee with respect to timing attacks.

**Optimized Implementation.** The performance of our optimized implementations on the aforementioned platform are described in Tables 10 – 13. This optimized implementation has been written in a constant time way whenever relevant thus its execution is expected to not leak any sensitive data.

**Known Answer Test Values.** 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.

Instance	Keygen	Sign	Verify
Mirath-1a-short	0.1 M	16.2 M	13.9 M
Mirath-1a-fast	0.1 M	5.9 M	3.3 M
Mirath-3a-short	0.2 M	133.3 M	130.4 M
Mirath-3a-fast	0.2 M	26.9 M	20.1 M
Mirath-5a-short	0.4 M	132.0 M	118.6 M
Mirath-5a-fast	0.4 M	40.1 M	28.4 M

**Table 10.** Performances of Mirath-a in Millions of CPU Cycles (Optimized implementation with Rijndael-based commitments)

Instance	Keygen	Sign	Verify
Mirath-1a-short	0.1 M	30.8 M	28.4 M
Mirath-1a-fast	0.1 M	8.5 M	6.0 M
Mirath-3a-short	0.2 M	143.9 M	140.1 M
Mirath-3a-fast	0.2 M	28.9 M	22.0 M
Mirath-5a-short	0.3 M	179.8 M	166.2 M
Mirath-5a-fast	0.3 M	48.4 M	36.6 M

**Table 11.** Performances of Mirath-a in Millions of CPU Cycles (Optimized implementation with SHA3-based commitments)

Instance	Keygen	Sign	Verify
Mirath-1b-short	0.6 M	23.6 M	18.3 M
Mirath-1b-fast	0.5 M	9.8 M	5.5 M
Mirath-3b-short	1.2 M	112.1 M	96.2 M
Mirath-3b-fast	1.2 M	47.2 M	38.2 M
Mirath-5b-short	1.9 M	155.3 M	131.6 M
Mirath-5b-fast	2.0 M	70.1 M	51.7 M

**Table 12.** Performances of Mirath-b in Millions of CPU Cycles (Optimized implementation with Rijndael-based commitments)

Instance	Keygen	Sign	Verify
Mirath-1b-short	0.6 M	37.5 M	33.0 M
Mirath-1b-fast	0.5 M	12.4 M	8.0 M
Mirath-3b-short	1.2 M	122.7 M	106.2 M
Mirath-3b-fast	1.2 M	49.4 M	39.9 M
Mirath-5b-short	2.0 M	201.2 M	178.0 M
Mirath-5b-fast	2.0 M	78.0 M	59.4 M

**Table 13.** Performances of Mirath-b in Millions of CPU Cycles (Optimized implementation with SHA3-based commitments)

## 7 Security Analysis

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  $(t, \epsilon_{PRG})$ -secure, and  $\epsilon_{SMR}$  the advantage an adversary has over the Syndrome MinRank problem. Consider  $\text{Hash}_1, \text{Hash}_2, \text{Hash}_3$  behave as random oracles, with an output of  $2\lambda$  bits. If an adversary makes  $q_i$  queries to  $\text{Hash}_i$  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:*

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

where  $q' = \max(q_1, q_2, q_3)$  and  $\tau$  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  $\Pr_i[\text{Forge}]$  the probability of forgery when considering game  $i$ . The aim of the proof is to find an upper bound on  $\Pr_1[\text{Forge}]$ .

- **Game 1.** This is the interaction between  $\mathcal{A}$  and the real signature scheme. KeyGen generates  $(\mathbf{H}, \mathbf{y}, \mathbf{S}, \mathbf{C}')$  and  $\mathcal{A}$  receives  $(\mathbf{H}, \mathbf{y})$ .  $\mathcal{A}$  can make queries to each  $\text{Hash}_i$  independently, and can make signing queries. At the end of the attack,  $\mathcal{A}$  outputs a message/signature pair,  $(m, \sigma)$ . 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  $\text{Hash}_1$ , or  $\text{Hash}_2$ , or  $\text{Hash}_3$ , then, the forgery is not valid. The first step is to look at the number of times every  $\text{Hash}_i$  is called when calling the signing oracle. The signing oracle contains one call to  $\text{Hash}_1$  and  $\text{Hash}_2$ , and  $(\tau \cdot N + 1)$  to  $\text{Hash}_3$ . The number of queries to  $\text{Hash}_1$ , to  $\text{Hash}_2$ , or to  $\text{Hash}_3$  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  $\text{Hash}_i$ ,  $q' = \max\{q_1, q_2, q_3\}$ ,  $q_S$  is the number of queries to the signing oracle. We thus have:

$$|\Pr_1[\text{Forge}] - \Pr_2[\text{Forge}]| \leq \frac{3 \cdot (q' + (\tau \cdot N + 1) \cdot q_S)^2}{2 \cdot 2^{2\lambda}}$$

- **Game 3.** The attacker now fails if the inputs to any of the  $\text{Hash}_i$  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  $\text{Hash}_i$ . This means, we can bound this with:

$$|\Pr_2[\text{Forge}] - \Pr_3[\text{Forge}]| \leq \frac{q_S \cdot (q_S + q_1 + q_2 + q_3)}{2^{2\lambda}} \leq \frac{q_S \cdot (q_S + 3 \cdot q')}{2^{2\lambda}}$$

- **Game 4.** When signing a message  $m$ , we now replace  $h_{\text{sh}}$  and  $h_{\text{piop}}$  with uniformly distributed random values. We then compute the challenges  $\Gamma$  and  $\{i^{*(e)}\}_e$ , and the value  $v_{\text{grinding}}$ , by expanding them. There is a difference with **Game 3** during a signing query, if a query to  $\text{Hash}_1$  or  $\text{Hash}_2$  was previously made. However, this does not happen as **Game 3** would already abort due to salt collision which means that:

$$\Pr_4[\text{Forge}] = \Pr_3[\text{Forge}]$$

- **Game 5.** We replace each  $\text{commit}_{e,i}$ , and  $h_{\text{com}}$ , with a uniformly distributed random value. Since  $h_{\text{com}} = \text{Hash}_1(\{\text{commit}_{e,i}\}_{e,i})$ , there cannot be a collision on  $h_{\text{com}}$  as there is no collision on the commitments since  $\text{commit}_{e,i}$  is expanded using  $e$  and  $i$ . This then means that, to have a collision on two queries to  $\text{Hash}_3$ , the same salt must be used, which is already an invalid forgery from the previous games thus:

$$\Pr_5[\text{Forge}] = \Pr_4[\text{Forge}]$$

- **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  $\epsilon_{PRG}$  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:

$$|\Pr_6[\text{Forge}] - \Pr_5[\text{Forge}]| \leq \tau \cdot q_S \cdot \epsilon_{PRG}$$

- **Game 7.** Finally, we say that an execution with index  $(e^*, \text{ctr})$  of a query  $h_{\text{piop}} = \text{Hash}_2(\text{pk}, \text{salt}, \text{msg}, h_{\text{sh}}, (\alpha_{\text{mid}}^{(e)}, \alpha_{\text{base}}^{(e)})_{e \in [1, \dots, \tau]})$  allows to retrieve a correct witness if:
  - $h_{\text{com}}$  is a query to  $\text{Hash}_3$ , i.e.,  $h_{\text{com}} = \text{Hash}_3(\{\text{commit}_{e,i}\}_{e,i})$ ;
  - Each  $\text{commit}_{e,i}$  is a query to  $\text{Hash}_3$ , i.e.,  $\text{commit}_{e,i} = \text{Hash}_3(\text{seed}_{e,i}, i \cdot \tau + e, \text{salt})$ ;
  - $h_{\text{sh}}$  is the output of a query to  $\text{Hash}_1$ , i.e.,  $h_{\text{sh}} = \text{Hash}_1(h_{\text{com}}, (\mathcal{S}_{\text{aux}}^{(e)}, \mathcal{C}'_{\text{aux}}^{(e)})_{e \in [1, \dots, \tau]})$ ;
  - The matrices  $\mathcal{S}, \mathcal{C}'$  defined by  $\{\text{state}_i^{(e^*)}\}_{i \in [1, \dots, N]}$  form a correct Syndrome MinRank solution;
  - $v_{\text{grinding}} = \mathbf{0} \in \{0, 1\}^w$  and  $\pi_{\text{BAVC}} \neq \perp$ .

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

$$\Pr_7[\text{Solve}] \leq \epsilon_{SMR}$$

Finally, we need to look at the upper bound of  $|\Pr_7[\text{Forge} \cap \overline{\text{Solve}}]|$ . Solve does not happen here, meaning that, to have a forgery after a query to  $\text{Hash}_2$ ,  $\mathcal{A}$  has no choice but to cheat either on  $\mathbf{I}$  or on  $\{i^{*(e^*)}\}$ . For that, he can:

- Find matrices  $\tilde{\mathbf{S}}$  and  $\tilde{\mathbf{C}}'$  such that  $\mathbf{H} \text{vec}(\tilde{\mathbf{S}} \cdot [\mathbf{I}_r \parallel \tilde{\mathbf{C}}']) \neq \mathbf{y}$  but such that  $\mathbf{I} \cdot (\mathbf{H} \text{vec}(\tilde{\mathbf{S}} \cdot [\mathbf{I}_r \parallel \tilde{\mathbf{C}}']) - \mathbf{y}) = \mathbf{0}$ , which happens with probability  $p = \frac{1}{q^{\mu \cdot \rho}}$ ;
- Successfully cheat on the polynomial  $P_\alpha$ , which happens with probability  $\frac{2}{N}$  because there are 2 roots to this polynomial.

Cheating on the second round must happen on the  $\tau$  repetitions thus the cheating probability is bounded by  $q' \cdot (\frac{2}{N})^\tau + q' \cdot \tau \cdot \frac{1}{q^{\mu \cdot \rho}}$ . In addition, because  $v_{\text{grinding}}$  must be equal to  $\mathbf{0}$ , the adversary  $\mathcal{A}$  has a success probability  $2^{-w}$  for each iteration  $(e^*, \text{ctr})$ . This results in a probability to cheat bounded by  $2^{-w} \cdot q' \cdot (\frac{2}{N})^\tau + q' \cdot \tau \cdot \frac{1}{q^{\mu \cdot \rho}}$ . Finally, the adversary must have  $\pi_{\text{BAVC}} \neq \perp$ . Let  $\theta$  be the probability that  $\pi_{\text{BAVC}} = \perp$  namely the probability that the sibling path exceeds the threshold  $T_{\text{open}}$ . This reduces the number of possible challenges from  $N^\tau$  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  $\theta$ . As a result, the adversary can cheat with probability  $\frac{2^\tau}{(N^\tau) \cdot (1 - \theta)} \cdot (1 - \theta) = (\frac{2}{N})^\tau$ .

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  $\tau$  challenge  $\tau'$  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  $\rho$  times, and  $P_2$  corresponds to the probability of cheating on the polynomial  $P_\alpha$  (see Section 3.1). Therefore, the KZ attack complexity is given by

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

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  $(\alpha_1, \dots, \alpha_k)$ . The second one, introduced by Bardet et al. [6], guesses  $a$  vectors (with a specific structure) in the right kernel of the secret  $\mathbf{E}$ . For each guess of the combined approach one derives a MinRank instance  $\tilde{\mathbf{M}}_0, \dots, \tilde{\mathbf{M}}_k$  with parameters  $(q, m, n - a, k - am - l_v, r)$ . Hence the complexity of this hybrid

approach is given by

$$q^{a \cdot r + l_v} \left( \text{MR\_Complexity}(q, m, n - a, k - am - l_v, r) + (\min\{k, am\})^\omega \right), \quad (4)$$

where  $\text{MR\_Complexity}(\cdot)$  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  $\mathbf{E}$  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  $\mathbf{M}_0, \dots, \mathbf{M}_k \in \mathbb{F}_q^{m \times n}$  and check if it has rank at most  $r$ . The complexity of this attack is  $\mathcal{O}(q^k \cdot r^\omega)$  [14].

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

*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  $\mathbf{E}$ . The expected complexity of this algorithm is

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

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

*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  $(\alpha_1, \dots, \alpha_k) \in \mathbb{F}_q^k$  and a matrix  $\mathbf{K} \in \mathbb{F}_q^{r \times (n-r)}$  such that

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

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

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

<sup>3</sup> 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  $\alpha_i$ 's and the entries of the matrix  $\mathbf{K}$ . 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  $\mathbf{M}$ . This is a determinantal system, whose Hilbert series is given by

$$HS(t) := \left[ (1-t)^{(m-r)(n-r)-(k+1)} \frac{\det(A(t))}{t^{\binom{r}{2}}} \right],$$

with  $A(t) = \left( \sum_{\ell=0}^{\max(m-i, n-j)} \binom{m-i}{\ell} \binom{n-j}{\ell} t^\ell \right)_{1 \leq i \leq r, 1 \leq j \leq r}$

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 sub-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  $\mathbf{M}'$  of  $\mathbf{M}$ , that has rank  $r$  and  $n' \geq r + 1$  columns, to introduce a formal matrix  $\mathbf{C} \in \mathbb{F}_q^{r \times n'}$  representing a generator matrix of this vector space. As  $\mathbf{C}$  is a basis for the rows of  $\mathbf{M}'$ , each row  $m_i$  of  $\mathbf{M}'$  is a linear combination of the rows of  $\mathbf{C}$ . This leads to the algebraic system

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

Considering the maximal minors of  $\mathbf{C}$  as new variables, the system consists of bilinear equations in the  $\alpha_i$ 's and the minors of  $\mathbf{C}$ . 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  $b \leq \min(q-1, r+1, n'-r)$  in the  $\alpha_i$ 's and degree one in

the minors of  $\mathbf{C}$ . 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

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

**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  $\mathbf{K}$  from the Kipnis-Shamir modeling (see Eq. (5)).

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

By the Kipnis-Shamir modeling, it holds that

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

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

Let  $\mathcal{S}$  be the set of all possible  $q^{rt}$  matrices of the form  $B(\mathbf{x}_1, \dots, \mathbf{x}_t) \in \mathbb{F}_q^{(k+1) \times mt}$ , with  $\mathbf{x}_1, \dots, \mathbf{x}_t \in \mathbb{F}_q^r$ . For the algorithm to work, we need to execute it using a value  $t$  such that  $B(\mathbf{k}_1, \dots, \mathbf{k}_t)$  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

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

operations over  $\mathbb{F}_q$ .

Determining the minimum  $t$  such that  $B(\mathbf{k}_1, \dots, \mathbf{k}_t)$  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  $\mathcal{O}(q^{-(mt-k)})$ , 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  $B(\mathbf{k}_1, \dots, \mathbf{k}_t)$  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 14.

*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  $2^{\text{maxdepth}}$  with  $\text{maxdepth} \leq 96$ . A parameter set is said to match the quantum security definition for a category if an attack requires at least  $2^{b-\text{maxdepth}}$  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  $D > 2^{\text{maxdepth}}$ , 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  $D_P = 2^{\text{maxdepth}}$  for a choice of  $P = (D/2^{\text{maxdepth}})^2$ .

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

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

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

In Table 14, 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 14.** 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 3.0 kB for ( $q = 2$ ) and 3.2 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]	$\sigma$ 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 15.** Keys and signature sizes of **Mirath-v-a** (VOLE variant,  $q = 16$ )

Instance	sk size [B]	pk size [B]	$\sigma$ 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 16.** 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 24, and the corresponding algorithms for key decompression are Algorithms 25 and 26, which in turn rely on the three subroutines described by Algorithms 27, 28, and 29.

Algorithm 24 is based on the method of Di Scala and Sanna [16] and generates a public key of the form  $\mathbf{pk} = (\text{seed}_{\mathbf{pk}}, \mathbf{y}_B)$ , where  $\text{seed}_{\mathbf{pk}}$  is a seed to generate  $\mathbf{H}'$  and  $\mathbf{y}_A$ . Hence, the size of the public key is

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

bits. A comparison of the public-key size of KeyGen and KeyGen2 for the parameter sets of Mirath is provided in Table 17.

Instance	public-key size [B]	
	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 17.** Public-key sizes of KeyGen and KeyGen2.

**Algorithm 24** Routine KeyGen2.

Mirath.KeyGen2()

```

1 : for  $t$  from 1 to max_tries do
2 :   seedpk  $\leftarrow$  TRG.Seed()
3 :   seedsk  $\leftarrow$  TRG.Seed()
4 :   ( $M_A$  [],  $H_B$  [])  $\leftarrow$  ExpandSeedMAHB(seedpk)
       $\triangleright M_A[i] \in \mathbb{F}_q^{m \times r}$  ( $i = 0, \dots, k$ ),  $H_B[j] \in \mathbb{F}_q^{m_0 \times 1}$  ( $j = 1, \dots, k$ )
5 :    $C' \leftarrow$  ExpandSeedC(seedsk)  $\triangleright C' \in \mathbb{F}_q^{r \times (n-r)}$ 
6 :    $\alpha \leftarrow$  GetSpecialSolution( $M_A$  [],  $C'$ )  $\triangleright \alpha \in \mathbb{F}_q^{k \times 1}$  or  $\alpha = \perp$ 
7 :   if  $\alpha \neq \perp$  then
8 :      $S \leftarrow M_A[0] + \sum_{i=1}^k \alpha_i M_A[i]$   $\triangleright S \in \mathbb{F}_q^{m \times r}$ 
9 :      $H_B[0] \leftarrow$  first  $m_0$  entries of  $\text{vec}(SC')$   $\triangleright H_B[0] \in \mathbb{F}_q^{m_0 \times 1}$ 
10 :     $y_B \leftarrow H_B[0] - \sum_{i=1}^k \alpha_i H_B[i]$   $\triangleright y_B \in \mathbb{F}_q^{m_0 \times 1}$ 
11 :    pk  $\leftarrow$  (seedpk,  $y_B$ )
12 :    sk  $\leftarrow$  (seedsk, seedpk)
13 :    return (pk, sk)
14 :   endif
15 : endfor
16 : return  $\perp$ 

```

**Algorithm 25** Routine DecompressPK2.

Mirath.DecompressPK2(pk)

```

1 : (seedpk,  $y_B$ )  $\leftarrow$  pk
2 : ( $M_A$  [],  $H_B$  [])  $\leftarrow$  ExpandSeedMAHB(seedpk)
       $\triangleright M_A[i] \in \mathbb{F}_q^{m \times r}$  ( $i = 0, \dots, k$ ),  $H_B[j] \in \mathbb{F}_q^{m_0 \times 1}$  ( $j = 1, \dots, k$ )
3 :  $H' \leftarrow - \begin{pmatrix} \text{vec}(M_A[1]) & \dots & \text{vec}(M_A[k]) \\ H_B[1] & \dots & H_B[k] \end{pmatrix}$   $\triangleright H' \in \mathbb{F}_q^{(mn-k) \times k}$ 
4 :  $y \leftarrow \begin{pmatrix} \text{vec}(M_A[0]) \\ y_B \end{pmatrix}$ 
5 : return ( $H'$ ,  $y$ )

```



---

**Algorithm 26** Routine DecompressSK2.

---



---

Mirath.DecompressSK2(sk)

---

```

1 : seedpk, seedsk ← sk
2 : (MA [], HB []) ← ExpandSeedMAHB(seedpk)
   ▷ MA[i] ∈  $\mathbb{F}_q^{m \times r}$  (i = 0, ..., k), HB[j] ∈  $\mathbb{F}_q^{m_0 \times 1}$  (j = 1, ..., k)
3 : C' ← ExpandSeedC(seedsk) ▷ K ∈  $\mathbb{F}_q^{r \times (n-r)}$ 
4 : α ← GetSpecialSolution(MA [], C') ▷ α ∈  $\mathbb{F}_q^{k \times 1}$  or α = ⊥
5 : if α = ⊥ then return ⊥
6 : S ← MA[0] + ∑i=1k αi MA[i]
7 : HB[0] ← first m0 entries of vec(SC') ▷ HB[0] ∈  $\mathbb{F}_q^{m_0 \times 1}$ 
8 : yB ← HB[0] − ∑i=1k αi HB[i] ▷ yB ∈  $\mathbb{F}_q^{m_0 \times 1}$ 
9 : H' ← −  $\begin{pmatrix} \text{vec}(M_A[1]) & \cdots & \text{vec}(M_A[k]) \\ H_B[1] & \cdots & H_B[k] \end{pmatrix}$  ▷ H' ∈  $\mathbb{F}_q^{(mn-k) \times k}$ 
10 : y ←  $\begin{pmatrix} \text{vec}(M_A[0]) \\ y_B \end{pmatrix}$ 
11 : return (H', y, S, C')
```

---



---

**Algorithm 27** Routine ExpandSeedMAHB.

---



---

Mirath.ExpandSeedMAHB(seed<sub>pk</sub>)

---

```

1 : prg ← PRG.Init(seedpk)
2 : for i from 0 to k do
3 :   MA[i] ← PRG.Sample(prg,  $\mathbb{F}_q^{m \times r}$ )
4 : endfor
5 : for i from 1 to k do
6 :   HB[i] ← PRG.Sample(prg,  $\mathbb{F}_q^{m_0 \times 1}$ )
7 : endfor
8 : return (MA [], HB [])
```

---



---

**Algorithm 28** Routine ExpandSeedC.

---



---

Mirath.ExpandSeedC(seed<sub>sk</sub>)

---

```

1 : prg ← PRG.Init(seedsk)
2 : C' ← PRG.Sample(prg,  $\mathbb{F}_q^{r \times (n-r)}$ )
3 : return C'
```

---

---

**Algorithm 29** Routine GetSpecialSolution.

---



---

GetSpecialSolution( $M_A \square, C'$ )

---

```

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  $\text{vec}(M_{A,j}C')$ 
         $\triangleright$  Note: no need to compute the full product  $M_{A,j}C'$ , since only one entry is needed
4 :   endfor
5 : endfor
6 :  $A \leftarrow I - (a_{i,j})_{1 \leq i,j \leq k} \triangleright A \in \mathbb{F}_q^{k \times k}$ 
7 :  $b \leftarrow (a_{i,0})_{1 \leq i \leq k} \triangleright b \in \mathbb{F}_q^{k \times 1}$ 
8 : Compute  $\alpha \in \mathbb{F}_q^{k \times 1}$  as the unique solution to the linear system  $Ax = b$ ,
   if such a solution exists unique, otherwise return  $\perp$ 
9 : return  $\alpha$ 

```

---

## 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. Zveydinger, and E. Bellini, *SoK: Cryptographic Estimators - 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  $F_5$  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.