MIRA Specifications

Nicolas Aragon[1], Magali Bardet[2,3], Loïc Bidoux[4], Jesús-Javier Chi-Domínguez[4], Victor Dyseryn[5], Thibauld Feneuil[6,7], Philippe Gaborit[5], Romaric Neveu[5], Matthieu Rivain[7], Jean-Pierre Tillich[3]

[1] Naquidis Center, Talence, France
[2] LITIS, University of Rouen Normandie, France
[3] INRIA, Paris, France
[4] Technology Innovation Institute, UAE
[5] University of Limoges, France
[6] Sorbonne Université, CNRS, INRIA, Institut de Mathématiques de Jussieu-Paris Rive Gauche, Ouragan, Paris, France
[7] Cryptoexperts, Paris, France

Table of Contents

1 Introduction

MIRA is a digital signature protocol based on the MinRank problem and the MPC-in-the-Head paradigm. This document specifies the scheme described in our Design Document, [ABCD$^+$23]. The scheme is quantum-resistant, and is based on zero-knowledge proofs and symmetric functions, such as hash functions. We present an high-level description of the scheme and the MPC-in-the-Head paradigm, on which the protocol is built. The security of the signature is conditioned by the difficulty of solving the MinRank problem. We apply the Fiat-Shamir transform [FS87] in order to have a signature. MIRA is based on additive secret sharing, and uses the idea of the hypercube structure from [AMGH$^+$22], as well as an MPC protocol using linearized polynomials which is introduced in [Fen22].

We propose two versions of the scheme (the only difference is the number of parties we simulate). One uses less parties and is faster but larger, while the other uses more parties and is shorter but slower.

The name MIRA comes from the MInRAnk problem.

2 Mathematical Background and Notations

To understand the mathematical background to the problem, we begin by recalling some fundamental definitions and we fix some notations.

We denote by $[1, N]$ the set of integer between 1 and N. This set can be shortened by writing $[N]$.
Let $\boldsymbol{E} \in \mathbb{F}_q^{m\times n} = \left(e_{i,j}\right), e_{i,j} \in \mathbb{F}_q, i \in [1, m], j \in [1, n]$, and $\mathcal{B} = \langle b_1 \dots b_m \rangle$ an $\mathbb{F}_q$-basis of $\mathbb{F}_{q^m}$. It is then possible to associate each column of $\boldsymbol{E}$ to an element of $\mathbb{F}_{q^m}$ by setting

$$e_j = \sum_{i=1}^{m} b_i e_{i,j}$$

for each $j \in [1, n]$
By setting $\boldsymbol{e} = (e_1 \dots e_n)$, we can say that $\boldsymbol{e}$ is the vector associated to the matrix $\boldsymbol{E}$.
The Rank Weight is defined as $\mathrm{W}_R(\boldsymbol{e}) = \mathsf{Rank}(\boldsymbol{E})$.
The distance between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is $d(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{W}_R(\boldsymbol{x} - \boldsymbol{y})$.
The support of $\boldsymbol{e} = (e_1 \dots e_n)$ is the linear subspace of $\mathbb{F}_{q^m}$ generated by its coordinates: $\mathsf{Supp}(\boldsymbol{e}) = \langle e_1 \dots e_n \rangle$.

With these notations in mind, we can define the MinRank Problem, on which MIRA is based:

MinRank:
Let $\mathbb{F}_q$ be the finite field of size q, and $m, n, k, r \in \mathbb{N}^*$. The computational MinRank Problem with parameters (q, m, n, k, r) is
Let $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k, \boldsymbol{E} \in \mathbb{F}_q^{m\times n}$ and $\boldsymbol{x} \in \mathbb{F}_q^k$ sampled uniformly at random such that:
$\boldsymbol{M}_0 = \boldsymbol{E} - \sum_{i=1}^{k} \boldsymbol{M}_i x_i$ and $\mathsf{Rank}(\boldsymbol{E}) \leq r$.
Knowing $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k$, retrieve $\boldsymbol{x}$.

The public key is the list of matrices $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k$, and the secret key is the vector $\boldsymbol{x}$. The principle of the signature is based on a transformation of a proof of knowledge of $\boldsymbol{x}$ into a non-interactive protocol thanks to the Fiat-Shamir Transform. Iterating the process multiple times gives the verifier a high assurance that the verifier knows $\boldsymbol{x}$.

The interactive proof relies on a prover simulating a MultiParty Computation (MPC) protocol, where each party has a share of the witness $\boldsymbol{x}$. The sharing of a secret s among N parties is denoted $(s[1], \ldots, s[N])$ where $s[i]$ is the share possessed by the party i. $s[J]$ where $J \subset [1, N]$ is the subset of shares $(s[j])_{j\in J}$. When we do not specify which party receives the share (as is the case in Fig.1), we note the share as $s[\cdot]$

Below is the secret sharing scheme we are going to use.

Additive Secret Sharing. Let $\mathbb{F}$ a field and $s \in \mathbb{F}$ a secret.
An additive secret sharing with N parties satisfies:
$s[i] = r_i$ for $i \in [1, N-1]$, where $r_i \xleftarrow{\$} \mathbb{F}$
$s[N] = s - \sum_{i=1}^{N-1} s[i]$
The $\mathsf{Reconstruct}_{[1,N]}$ algorithm takes as inputs all the shares, and outputs the sum of all the shares.

3 High-level description of the signature scheme

As mentioned earlier, the scheme relies on a MinRank instance. Hence, the public key pk is a list of matrices $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k$, and the private key sk is the vector $\boldsymbol{x}$, of size k.
In order to sign a message one has to:

– Build the proof of knowledge;

– Apply the Fiat-Shamir transform.

The proof of knowledge relies on the MPC protocol of [Fen22] (see [ABCD$^+$23]).

3.1 Description of the MPC protocol

A multi-party computation protocol is an interactive protocol involving several parties whose objective is to jointly compute a function f on the shares of x they received at the begin of the protocol, and each get a share of $f(x)$. At each step, the parties can perform one of the following actions:

- Receiving elements: it can be randomness sent by a random oracle, or the share of an hint which depends on the witness w and the previous elements sent;
- Computing: since the sharing is linear, the parties can perform linear transformations on their shares;
- Broadcasting: the parties can broadcast their shares of a given value which is then publicly recomputed by all the parties. This opening should be done in a way that do not leak information about the witness w.

Due to application to zero-knowledge proofs as detailed below, the MPC protocol only needs to be secure in the semi-honest model in which all the parties honestly follow the different steps of the protocol.

The MPC protocol we use here allows N parties to verify that a matrix, $\boldsymbol{E}$, built using the secret $\boldsymbol{x}$, is of a certain rank. It works this way:

- Each party receives their shares of the witness $\boldsymbol{x}$, as well as shares of some elements that will be used to execute the protocol;
- Each party computes their share of $\boldsymbol{E}$;
- After some other computations, each party computes a value v, which is a combination of evaluations of a linearized polynomial. If $v = 0$, $\boldsymbol{x}$ is the witness corresponding to the MinRank instance;
- The view of $N - 1$ parties gives no information whatsoever about $\boldsymbol{x}$.

Concretely, if $v = 0$, this means that $\boldsymbol{E}$ is of rank at most r if the computations and the inputs are from an honest party. This comes from the properties of a linearized polynomial (see [ABCD$^+$23] or [Fen22]).
Formally, we can describe the MPC protocol with the following figure:

```
- Public data $M_0, \ldots, M_k \in \mathbb{F}_q^{m \times n}$
- Each party receives $x[\cdot] \in \mathbb{F}_q^k$, $L[\cdot] = \sum_{i=0}^{r-1} \beta_i[\cdot] X^{q^i} + X^{q^r}$ where each $\beta_i \in \mathbb{F}_{q^m}$, $a[\cdot]$ where $a$ is uniformly sampled from $\mathbb{F}_{q^{m\eta}}^r$ and $c[\cdot] \in \mathbb{F}_{q^{m\eta}}$ such that $c = -\langle a, \beta \rangle$
1. The parties sample $(\gamma_1, ..., \gamma_n, \epsilon) \xleftarrow{\$} \mathbb{F}_{q^{m\eta}}^{n+1}$
2. The parties compute $E[\cdot] = M_0 + \sum_{i=1}^k x_i[\cdot] M_i$
3. The parties set $e_i[\cdot]$ as the element in $\mathbb{F}_{q^m}$ associated to the i-th column of $E[\cdot]$
4. They compute $z[\cdot] = -\sum_{j=1}^n \gamma_j (e_j[\cdot]^{q^r})$
5. For $i$ from 0 to $r-1$ :
    ◇ They compute $w_i[\cdot] = \sum_{j=1}^n \gamma_j (e_j[\cdot]^{q^i})$
6. $\alpha[\cdot] = \epsilon \cdot w[\cdot] + a[\cdot]$ and the parties open the shares to get $\alpha$
7. $v[\cdot] = \epsilon \cdot z[\cdot] - \langle \alpha, \beta[\cdot] \rangle - c[\cdot]$ and the parties open the shares to get $v$
8. They output ACCEPT if $v = 0$, and REJECT otherwise
```

Fig. 1: Protocol Π^η to check that an input is solution of an instance of MinRank

Steps 2 and 3 are the computation of $\boldsymbol{E}$ and its support, and steps 4 to 7 are the computation of v. The parties open an element, $\boldsymbol{\alpha}$, during the process, in order to make sure the computation is done honestly. Note that in step 3, the computation is done in regards to the columns of $\boldsymbol{E}$. Depending on the parameters m and n, it might be better to do it with the rows instead.
We refer to [ABCD$^+$23] or [Fen22] for the proof of false positive of this MPC protocol and its correctness.

3.2 Application of the MPCitH paradigm

3.2.1 General view of MPCitH

The zero-knowledge proof relies on the application of the MPCitH paradigm [IKOS07].

The MPCitH paradigm consists in committing the views of all the parties, and revealing as much as possible without leaking information about $\boldsymbol{x}$. Generally, an MPCitH protocol works as follows:

- The prover commits the shares of the parties;
- He receives a challenge in order to execute the MPC protocol;
- He sends the hash of the results of the MPC protocol for each party;
- He receives a challenge to know what to reveal;
- He reveals what he has to according to the second challenge.

We refer to [FR22] for a more formal definition.

We will now describe the proof of knowledge.

3.2.2 Overview of MIRA

The hypercube structure is used in this case. The idea, introduced in [AMGH$^+$22], is the following:
The prover builds $N = 2^D$ shares, using additive sharing, and organises them in a hypercube with D dimensions. Then, it is possible to sum shares, in order to get 2 *main shares* per dimension only. Since it is an additive scheme, all that is left to do for him is to execute the MPC protocol D times on the 2 *main shares* of the dimension. Moreover, since the D MPC protocols use the same secret, one can get further savings by only performing the MPC computation for one main share per dimension as well as the MPC computation on the plain values. This makes a total of $D + 1$ MPC computations instead of 2^D. The interested reader can find more details in our design documentation [ABCD$^+$23] and in [AMGH$^+$22] regarding the hypercube approach. We obtain the following zero-knowledge proof of knowledge protocol:

- The prover builds the inputs of the MPC protocol, i.e, he simulates the states of N parties;
- He then commits the states of the parties, computes the *main shares*, and receives a first challenge corresponding to some random values;
- Thanks to this challenge, he simulates the $D - 1$ MPC computations on main shares (one per dimension), and 1 MPC computation on plain values, and sends the hash of the results of the computations;
- He receives a second challenge, which is a subset of $N - 1$ parties, and reveals their views. (In practice, he receives only a subset of 1 party, and reveals the views of every party except this one). He also reveals one the broadcast shares of the party whose view is not revealed;
- The verifier checks that the computations of the MPC protocol was done honestly and that the MPC protocol would accept the shared input solution;

This zero-knowledge proof of knowledge can then be turned into a signature scheme using the Fiat-Shamir transform [FS87].

4 Detailed algorithmic description

This section is dedicated to a low-level algorithmic description of the three algorithms of our scheme: `MIRA_Keygen`, `MIRA_Sign` and `MIRA_Verify`. Before that, we introduce low-level notation in Section 4.1, and detail subroutines of our main algorithmis in Sections 4.2-4.8.

The MPC protocol from Figure 1 is characterized by a degree η of extension of $\mathbb{F}_{q^m}$ in which the computations are done. The signature size is minimized for $\eta = 1$, as presented in Section 5. The rest of this section assumes $\eta = 1$ for simplification purposes.

4.1 Algorithmic notation

For an integer $x \in [0, 2^D - 1]$ and $\delta \in [D]$, we denote $\texttt{bit}(x, \delta)$ the δ-th least significant bit of the binary repesentation of x. For example, let $x = 35 = 100011_2$, then $\texttt{bit}(x, 2) = 1$ and $\texttt{bit}(x, 3) = 0$.

Indexes:		
i	$[1, N]$	Index of a leaf party.
δ	$[1, D]$	Index of a main party.
e	$[1, \tau]$	Index of an iteration.
Seeds:		
`seed_pk`	$\{0,1\}^\lambda$	Seed for generation of the matrix $\boldsymbol{H}$ in the public key.
`seed_sk`	$\{0,1\}^\lambda$	Seed for generation of the secret vector $\boldsymbol{x}$ and secret annihilator polynomial $\boldsymbol{\beta}$.
`mseed`	$\{0,1\}^\lambda$	Master seed for generation of the seeds $\texttt{seed}^{(e)}$
$\texttt{seed}^{(e)}$	$\{0,1\}^\lambda$	Root seed of the PRG tree.
$\texttt{seed}_i^{(e)}$	$\{0,1\}^\lambda$	Leaf seed of the PRG tree.
$\texttt{ptree}^{(e)}$	$\{0,1\}^{\lambda D}$	Partial seeds of the PRG tree masking an index $i^{*(e)}$.
Constants:		
`DS_M`	$\{0,1\}^\lambda$	Domain separator for the message.
`DS_T`	$\{0,1\}^\lambda$	Domain separator for the PRG tree.
`DS_C`	$\{0,1\}^\lambda$	Domain separator for the PRG commitments.
`DS_1`	$\{0,1\}^\lambda$	Domain separator for the first response.
`DS_2`	$\{0,1\}^\lambda$	Domain separator for the second response.
Bytestring variables:		
$\texttt{cmt}_i^{(e)}$	$\{0,1\}^{2\lambda}$	Leaf commitments.
h_1	$\{0,1\}^{2\lambda}$	First commitment.
h_2	$\{0,1\}^{2\lambda}$	Second commitment.
Field elements, vectors and matrices:		
$\boldsymbol{M}_i$	$\mathbb{F}_q^{m \times n}$	Public matrix
$\boldsymbol{E}$	$\mathbb{F}_q^{m \times n}$	Secret error matrix of rank r
$\boldsymbol{e}$	$\mathbb{F}_{q^m}^n$	$\mathbb{F}_{q^m}$-vector associated to the $\mathbb{F}_q$-matrix $\boldsymbol{E}$
$\boldsymbol{x}$	$\mathbb{F}_q^k$	Secret coefficients of the secret linear combination
$\boldsymbol{\beta}$	$\mathbb{F}_{q^m}^r$	Annulator polynomial of the support of $\boldsymbol{e}$.
$\boldsymbol{a}$	$\mathbb{F}_{q^m}^r$	Vector used in the MPC rank checking protocol.
$\boldsymbol{w}$	$\mathbb{F}_{q^m}^r$	Vector used in the MPC rank checking protocol.
$\boldsymbol{\gamma}$	$\mathbb{F}_{q^m}^n$	Vector from the first challenge.
c	$\mathbb{F}_{q^m}$	Element used in the MPC rank checking protocol.
z	$\mathbb{F}_{q^m}$	Element used in the MPC rank checking protocol.
ϵ	$\mathbb{F}_{q^m}$	Element from the first challenge.

The leaf shares of the above vectors and elements are noted with an array index i (for example $\boldsymbol{a}[i]$). The main shares of the above vectors and elements are noted with an hat and an array index δ (for example $\hat{\boldsymbol{a}}[\delta]$).

Table 1: Description of low level notation used in our scheme

4.2 Operations on finite field elements

Representation of finite field elements

In this document we always have $\mathbb{F}_q = \mathbb{F}_{16}$. We define $\mathbb{F}_q$ as $\mathbb{F}_2[X]/\langle P \rangle$ where $P = X^4 + X + 1$. Elements of $\mathbb{F}_q$ are represented in the polynomial basis.

Converting a byte stream into elements of $\mathbb{F}_q$

When sampling randomness, we want to convert a byte stream $b_0, b_1, \ldots$ into elements of $\mathbb{F}_q$. Sampling n elements in $\mathbb{F}_q$ is done using the `FqArrayFromBytes(n)` function 1.

Algorithm 1 `FqArrayFromBytes`

```
Input: Size $n$, $l = \lceil \frac{n}{2} \rceil$ bytes $b_0, \ldots, b_{l-1}$
Output: An array $\boldsymbol{x}$ of $n$ elements of $\mathbb{F}_q$
1: for $i$ from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ do
2:     $\boldsymbol{x}_{2i} = b_i$ & 0xf
3:     $\boldsymbol{x}_{2i+1} = (b_i >> 4)$
4: if $n\%2 == 1$ then
5:     $\boldsymbol{x}_{n-1} = b_{l-1}$ & 0xf
```

We also define the `FqArrayToBytes(A)` 2 to go from an array of n elements of $\mathbb{F}_q$ to a byte array.

Algorithm 2 `FqArrayToBytes`

```
Input: An array $\boldsymbol{x}$ of $n$ elements of $\mathbb{F}_q$
Output: A byte array $b_0, \ldots, b_{l-1}$ where $l = \lceil \frac{n}{2} \rceil$
1: for $i$ from 0 to $\lfloor \frac{n}{2} \rfloor - 1$ do
2:     $b_i = (\boldsymbol{x}_{2i}$ & 0xf$) + (\boldsymbol{x}_{2i+1} << 4)$
3: if $n\%2 == 1$ then
4:     $b_{l-1} = \boldsymbol{x}_{n-1}$ & 0xf
```

Using these core functions we can define how matrices over $\mathbb{F}_q$ and arrays of elements in $\mathbb{F}_{q^m}$ are parsed:

- We process arrays of r elements of $\mathbb{F}_{q^m}$ as arrays of rm elements of $\mathbb{F}_q$,
- We process $m \times n$ matrices over $\mathbb{F}_q$ as arrays of mn elements of $\mathbb{F}_q$.

We refer to the resulting functions as `FqmArrayFromBytes(size)`, `FqmArrayToBytes(`$\boldsymbol{x}$`)`, `FqMatrixFromBytes(lines, columns)` and `FqMatrixToBytes(M)`.

Defining extensions of $\mathbb{F}_q$

In order to perform operations in $\mathbb{F}_{q^m}$ we define how $\mathbb{F}_{q^m}$ is defined as an extension of $\mathbb{F}_q$.

- For $m = 16$:

  We define $\mathbb{F}_{q^2}$, $\mathbb{F}_{q^4}$, $\mathbb{F}_{q^8}$ and $\mathbb{F}_{q^{16}}$ as follows:

$$\mathbb{F}_{q^2} = \mathbb{F}_q[Z]/\langle Z^2 + Z + X^3 \rangle$$

$$\mathbb{F}_{q^4} = \mathbb{F}_{q^2}[A]/\langle A^2 + A + (ZX^3)\rangle$$
$$\mathbb{F}_{q^8} = \mathbb{F}_{q^4}[B]/\langle B^2 + B + (ZAX^3)\rangle$$
$$\mathbb{F}_{q^{16}} = \mathbb{F}_{q^8}[Y]/\langle Y^2 + Y + (ZABX^3)\rangle$$

– For m prime:

  When m is prime we define $\mathbb{F}_{q^{16}}$ as $\mathbb{F}_q[Y]/\langle P\rangle$ where P is an irreducible polynomial of degree m over $\mathbb{F}_2$. Table 2 describes the chosen polynomials.

m	P
19	$Y^{19} + Y^5 + Y^2 + Y + 1$
23	$Y^{23} + Y^5 + 1$

Table 2: Polynomials used to define the field $\mathbb{F}_{q^m}$.

Conversion from $\mathbb{F}_q$-arrays to $\mathbb{F}_{q^m}$-elements is done using a $\mathbb{F}_q$-basis of $\mathbb{F}_{q^m}$ $(g_1, ..., g_m)$ that is defined as follows:

– For $m = 16$, the basis consists of elements $Z^{b_0}A^{b_1}B^{b_2}Y^{b_3}$ for every possible 4-bitstring $b_0b_1b_2b_3$ in increasing order.

– For m prime, the basis is the standard polynomial basis $(1, Y, \ldots, Y^{m-1})$.

This basis $(g_1, ..., g_m)$ allows converting between an $\mathbb{F}_q$-matrix of size $m \times n$ and an $\mathbb{F}_{q^m}$-vector of size n:

Algorithm 3 `FqMatrixToFqmVector`

```
Input: A matrix $\boldsymbol{M} \in \mathbb{F}_q^{m \times n}$
Output: A vector $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$ of length $n$
1: for $j \in [n]$ do
2:     $\boldsymbol{x}_j = \sum_{i=1}^{m} \boldsymbol{M}_{i,j} g_i$
3: return $\boldsymbol{x}$
```

Algorithm 4 `FqmVectorToFqMatrix`

```
Input: A vector $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$ of length $n$
Output: A matrix $\boldsymbol{M} \in \mathbb{F}_q^{m \times n}$
1: for $j \in [n]$ do
2:     Fill the $j$-th column of $\boldsymbol{M}$ with the coefficients of $\boldsymbol{x}_j$ in the basis $(g_1, \ldots, g_m)$.
           $$\boldsymbol{x}_j = \sum_{i=1}^{m} \boldsymbol{M}_{i,j} g_i$$
3: return $\boldsymbol{M}$
```

Computing an annihilator polynomial

We use the procedure from [Loi07, Ch. 3, Sec. 2.4] to compute the coefficients of a linearized polynomial given its roots.

Algorithm 5 `ComputeAnnihilatorPolynomial`

Input: Support $\boldsymbol{supp} \in \mathbb{F}_{q^m}^r$ of rank r
Output: Annihilator polynomial $P \in \mathbb{F}_{q^m}[X]$ such that for all $u \in \langle \boldsymbol{supp} \rangle, P(u) = 0$

```
$P_1 = X^q - \boldsymbol{supp}[0]^{q-1}$
for $i$ from 1 to $r$ do
    $T = P_i^q$
    $eval = P_i(\boldsymbol{supp}[i])$
    $P_{i+1} = eval^{q-1} P_i + T$
return $P_{r+1}$
```

4.3 Randomness generators and hash functions

- `RandomBytes(ℓ)` is instanciated using the NIST provided `randombytes` function, it returns ℓ bytes from system entropy;
- `PRG` is a psuedorandom generator instanciated using `SHAKE-128` for security category 1 and `SHAKE-256` otherwise. It offers two invokable functions:
  - `PRG.Init(seed)` initializes the internal state of the generator with a seed,
  - `PRG.GetBytes(ℓ)` outputs ℓ bytes from the generator and updates its internal state.
- `Hash` functions are instanciated using `SHA3-256`, `SHA3-384` or `SHA3-512` for security categories 1, 3 and 5 respectively.

4.4 Sampling routines

These routines sample from a seed some set of elements needed in the scheme. Recall that we only consider the case where $q = 16$.

Algorithm 6 `SampleSecretInstance`

Input: Size (m, n), dimension k, rank weight r, a seed `seed`

Output: $\boldsymbol{E} \in \mathbb{F}_q m \times n$ a matrix of rank r and its support $\boldsymbol{supp} \in \mathbb{F}_{q^m}^r$, $\boldsymbol{x} \in \mathbb{F}_q^k$.

```
 1: PRG.Init(seed)
 2: r_bytes = $\lceil \frac{rm}{2} \rceil$
 3: repeat
 4:     $\boldsymbol{supp}[1, r]$ = FqmArrayFromBytes(r, PRG.GetBytes(r_bytes))     ▷ Sampling the support
 5: until Rank(FqmVectorToFqMatrix($\boldsymbol{supp}$)) = $r$
 6: $\boldsymbol{e} = (0, \ldots, 0)$                                                 ▷ Initialize $\boldsymbol{e}$ with zeros
 7: repeat
 8:     coordinates =FqArrayFromBytes(nr, PRG.GetBytes($\lceil \frac{nr}{2} \rceil$))
 9:     for $i \in [n]$ do
10:         for $j \in [r]$ do
11:             $\boldsymbol{e}_i = \boldsymbol{e}_i$ + coordinates$[(i-1)r + j] \cdot \boldsymbol{supp}[j]$
12:     $\boldsymbol{E}$ = FqmVectorToFqMatrix($\boldsymbol{e}$)
13: until Rank($\boldsymbol{E}$) = $r$
14: k_bytes = $\lceil \frac{k}{2} \rceil$
15: $\boldsymbol{x}$ = FqArrayFromBytes(k, PRG.GetBytes(k_bytes))
16: return $\boldsymbol{E}, \boldsymbol{supp}, \boldsymbol{x}$
```

Algorithm 7 `SampleSeeds`

Input: Length ℓ, Number of seeds τ, a seed `seed`

Output: $(\texttt{seed}^{(e)})_{e \in [\tau]} \in (\{0,1\}^{\ell})^{\tau}$

```
PRG.Init(seed)
for $i \in [\tau]$ do
    seed$^{(e)}$ = PRG.GetBytes($\ell$)
return (seed$^{(e)}$)$_{e \in [\tau]}$
```

Algorithm 8 `SampleShares`

Input: A seed `seed`

Output: A set of shares $(\boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{a}, c) \in \mathbb{F}_q^k \times \mathbb{F}_{q^m}^r \times \mathbb{F}_{q^m}^r \times \mathbb{F}_{q^m}$

```
PRG.Init(seed)
xbytes = PRG.GetBytes($\lceil \frac{k}{2} \rceil$)
$\boldsymbol{x}$ = FqArrayFromBytes(k, xbytes)
bbytes = PRG.GetBytes($\lceil \frac{rm}{2} \rceil$)
$\boldsymbol{\beta}$ = FqmArrayFromBytes(r, bbytes)
abytes = PRG.GetBytes($\lceil \frac{rm}{2} \rceil$)
$\boldsymbol{a}$ = FqmArrayFromBytes(r, abytes)
cbytes = PRG.GetBytes($\lceil \frac{m}{2} \rceil$)
$c$ = FqmArrayFromBytes(1, cbytes)[0]
return $(\boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{a}, c)$
```

Algorithm 9 `SampleFirstChallenge`

Input: A seed `seed`
Output: A set of first challenges $(\boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})_{e\in[\tau]} \in (\mathbb{F}_{q^m}^n \times \mathbb{F}_{q^m})^\tau$

```
PRG.Init(seed)
for $e \in [\tau]$ do
    xbytes $=$ PRG.GetBytes($\lceil \frac{(n+1)m}{2} \rceil$)
    $\boldsymbol{\gamma}^{(e)}$ $=$ FqmArrayFromBytes(n, xbytes[..n])
    $\epsilon^{(e)}$ $=$ FqmArrayFromBytes(1, xbytes[n+1])$[0]$
return $(\boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})_{e\in[\tau]}$
```

Algorithm 10 `SampleSecondChallenge`

Input: A seed `seed`
Output: A set of second challenges $(i^{*(e)})_{e\in[\tau]} \in [N]^\tau$

```
PRG.Init(seed)
for $e \in [\tau]$ do
    $i^{*(e)}$ $=$ PRG.GetBytes(1)   $\bmod N$
return $(i^{*(e)})_{e\in[\tau]}$
```

4.5 MPC routines

Algorithm 11 `MPC-MinRank.ComputeAlpha`

Input:

- A set of shares $(\boldsymbol{e}, \boldsymbol{a})$
- A protocol challenge $\big((\gamma_j)_{j\in[1,n]}, \epsilon\big)$

Output: $\boldsymbol{\alpha}$

```
1: for $k \in [1, r]$ do
2:     $w_k = \sum_{j=1}^{n} \gamma_j (\boldsymbol{e}_j^{q^{k-1}})$
3: $\boldsymbol{\alpha} = \epsilon \cdot \boldsymbol{w} + \boldsymbol{a}$
4: return $\boldsymbol{\alpha}$
```

Algorithm 12 `MPC-MinRank.Exec`

Input:

- A set of main shares $(\hat{\boldsymbol{x}}_B[\delta], \hat{\boldsymbol{\beta}}[\delta], \hat{\boldsymbol{a}}[\delta], \hat{c}[\delta])_{\delta \in [D]}$
- A protocol challenge $\big((\gamma_j)_{j \in [1,n]}, \epsilon\big)$
- A full value $\boldsymbol{\alpha}$
- The public matrices $\boldsymbol{M_0}, \ldots, \boldsymbol{M_k}$

Output: $(\hat{\boldsymbol{\alpha}}[\delta], \hat{v}[\delta])_{\delta \in [D]}$

```
1: for $\delta \in [D]$ do
2:     $\hat{\boldsymbol{E}}[\delta] = \boldsymbol{M}_0 + \sum_{i=1}^{k} \hat{x}_i[\delta] \boldsymbol{M}_i$
3:     $\hat{\boldsymbol{e}}[\delta] = \texttt{FqMatrixToFqmVector}(\hat{\boldsymbol{E}}[\delta])$
4:     $\hat{\boldsymbol{\alpha}}[\delta] = \texttt{MPC-RSD.ComputeAlpha}(\hat{\boldsymbol{e}}[\delta], \hat{\boldsymbol{a}}[\delta], (\gamma_j)_{j \in [1,n]}, \epsilon)$
5:     $\hat{z}[\delta] = -\sum_{j=1}^{n} \gamma_j (\hat{\boldsymbol{e}}_j[\delta]^{q^r})$
6:     $\hat{v}[\delta] = \epsilon \cdot \hat{z}[\delta] - \langle \boldsymbol{\alpha}, \hat{\boldsymbol{\beta}}[\delta] \rangle - \hat{c}[\delta]$
7: return $(\hat{\boldsymbol{\alpha}}[\delta], \hat{v}[\delta])_{\delta \in [D]}$
```

4.6 Hypercube routines

The $N = 2^D$ leaf shares are arranged on a D-dimensional hypercube of side length 2. An index $i \in [N]$ is positioned according to the binary representation of $i - 1 \in [0, 2^D - 1]$. The δ-th coordinate of i in the hypercube is the δ-th least significant bit of the binary repesentation of $i - 1$, i.e. $\texttt{bit}(i - 1, \delta)$.

A main share of index δ is the sum of all leaf shares in the hyperface of the hypercube comprising all vertices whose δ-th coordinate is zero.

Algorithm 13 `Hypercube.MainShares-Compute`

Input: A set of leaf shares $(\boldsymbol{x}[i], \boldsymbol{\beta}[i], \boldsymbol{a}[i], c[i])_{i \in [N]}$ where $N = 2^D$
Output: A set of main shares $(\hat{\boldsymbol{x}}[\delta], \hat{\boldsymbol{\beta}}[\delta], \hat{\boldsymbol{a}}[\delta], \hat{c}[\delta])_{\delta \in [D]}$

```
1: for $\delta \in [D]$ do
2:     $\hat{\boldsymbol{x}}[\delta] = \sum_{\substack{i \in [N] \\ \texttt{bit}(i-1,\delta)=0}} \boldsymbol{x}[i]$
3:     $\hat{\boldsymbol{\beta}}[\delta] = \sum_{\substack{i \in [N] \\ \texttt{bit}(i-1,\delta)=0}} \boldsymbol{\beta}[i]$
4:     $\hat{\boldsymbol{a}}[\delta] = \sum_{\substack{i \in [N] \\ \texttt{bit}(i-1,\delta)=0}} \boldsymbol{a}[i]$
5:     $\hat{c}[\delta] = \sum_{\substack{i \in [N] \\ \texttt{bit}(i-1,\delta)=0}} c[i]$
```

Algorithm 14 `Hypercube.MainAlphaAndV-Compute`

Input: A set of leaf shares $(\boldsymbol{\alpha}[i], v[i])_{i\in[N]}$ where $N = 2^D$
Output: A set of main shares $(\hat{\boldsymbol{\alpha}}[\delta], \hat{v}[\delta])_{\delta\in[D]}$

```
1: for $\delta \in [D]$ do
2:     $\hat{\boldsymbol{\alpha}}[\delta] = \sum_{\substack{i\in[N] \\ \texttt{bit}(i-1,\delta)=0}} \boldsymbol{\alpha}[i]$
3:     $\hat{v}[\delta] = \sum_{\substack{i\in[N] \\ \texttt{bit}(i-1,\delta)=0}} v[i]$
```

4.7 Key parsing routines

Algorithm 15 `ParseSecretKey`

Input: Secret key sk
Output: $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k, \boldsymbol{x}$, the annihilator polynomial $\boldsymbol{\beta}$ and $\boldsymbol{e}$

```
1: $(\texttt{sk\_seed}, \texttt{pk\_seed}) = \mathsf{sk}$
2: pk_PRG.Init(pk_seed)
3: $\texttt{L\_bytes} = \lceil \frac{k(mn-k)}{2} \rceil$
4: $\boldsymbol{L'}$ = FqMatrixFromBytes(k, mn-k, pk_PRG.GetBytes(L_bytes))
5: $\boldsymbol{L} = (I_k|\boldsymbol{L'})$
6: for $i \in [k]$ do
7:     for $j \in [m]$ do
8:         for $\ell \in [n]$ do
9:             $(\boldsymbol{M}_i)_{j,\ell} = \boldsymbol{L}_{i,(j-1)n+\ell}$
10: $(\boldsymbol{E}, \boldsymbol{supp}, \boldsymbol{y})$ = SampleSecretInstance(sk_seed)
11: $\boldsymbol{F} = \boldsymbol{E} - \sum_{i=1}^{k} \boldsymbol{y}_i \boldsymbol{M}_i$
12: for $i \in [k]$ do
13:     $f_i = \boldsymbol{F}_{1+\lfloor \frac{i-1}{n} \rfloor, 1+((i-1) \mod n)}$
14: $\boldsymbol{M}_0 = \boldsymbol{F} - \sum_{i=1}^{k} f_i \boldsymbol{M}_i$
15: $\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{f}$
16: $\boldsymbol{e}$ = FqMatrixToFqmVector($\boldsymbol{E}$)
17: $\boldsymbol{\beta}$ = ComputeAnnihilatorPolynomial($\boldsymbol{supp}$)
18: return $(\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{e})$
```

Algorithm 16 `ParsePublicKey`

Input: Public key `pk`
Output: $\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k$

```
 1: $(\texttt{pk\_seed}, \texttt{m\_bytes}) = \texttt{pk}$
 2: PRG.Init(pk_seed)
 3: $\texttt{L\_bytes} = \lceil \frac{k(mn-k)}{2} \rceil$
 4: $\boldsymbol{L}' = \texttt{FqMatrixFromBytes(k, mn-k, PRG.GetBytes(L\_bytes))}$
 5: $\boldsymbol{L} = (I_k | \boldsymbol{L}')$
 6: for $i \in [k]$ do
 7:     for $j \in [m]$ do
 8:         for $\ell \in [n]$ do
 9:             $(\boldsymbol{M}_i)_{j,\ell} = \boldsymbol{L}_{i,(j-1)n+\ell}$
10: $\boldsymbol{M}_0 = \texttt{FqMatrixFromBytes(m, n, m\_bytes)}$
11: return $(\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k)$
```

4.8 PRG tree routines

Algorithm 17 `PRGTreeExpand`

Input: A root seed `seed`, a number of parties $N = 2^D$, a salt and an iteration index e
Output: A family of seeds $(\texttt{seed}_i)_{i \in [N]}$

```
1: if $D = 0$ then
2:     return seed
3: else
4:     $(\texttt{seed\_left}, \texttt{seed\_right}) = \texttt{Hash}(\texttt{DS\_T}, \mathsf{salt}, e, \texttt{seed})$
5:     $(\texttt{seed}_i)_{i \in [1, N/2]} = \texttt{PRGTreeExpand}(\texttt{seed\_left}, N/2, \mathsf{salt}, e)$
6:     $(\texttt{seed}_i)_{i \in [N/2+1, N]} = \texttt{PRGTreeExpand}(\texttt{seed\_right}, N/2, \mathsf{salt}, e)$
7:     return $(\texttt{seed}_i)_{i \in [N]}$
```

Algorithm 18 `PRGPartialTreeReveal`

Input: A root seed $\texttt{seed}$, a number of parties $N = 2^D$, a salt and an iteration index e, a hidden index i^*
Output: A partial tree, i.e. a family of seeds $(\texttt{seed}_i)_{i\in[D]}$

```
 1: $(\texttt{seed\_left}, \texttt{seed\_right}) = \texttt{Hash}(\texttt{DS\_T}, \mathsf{salt}, e, \texttt{seed})$
 2: if $D = 1$ then
 3:     if $i^* = 1$ then
 4:         return $\texttt{seed\_right}$
 5:     else
 6:         return $\texttt{seed\_left}$
 7: else
 8:     if $i^* \leq N/2$ then
 9:         $(\texttt{seed}_i)_{i\in[1,D-1]} = \texttt{PRGPartialTreeReveal}(\texttt{seed\_left}, N/2, \mathsf{salt}, e, i^*)$
10:         $\texttt{seed}_D = \texttt{seed\_right}$
11:     else
12:         $\texttt{seed}_1 = \texttt{seed\_left}$
13:         $(\texttt{seed}_i)_{i\in[2,D]} = \texttt{PRGPartialTreeReveal}(\texttt{seed\_right}, N/2, \mathsf{salt}, e, i^* - N/2)$
14:     return $(\texttt{seed}_i)_{i\in[D]}$
```

Algorithm 19 `PRGPartialTreeExpand`

Input: A partial tree, i.e. a family of seeds $(\texttt{seed}_i)_{i\in[D]}$, a salt and an iteration index e, a hidden index i^*
Output: All seeds but one $(\texttt{seed}_i)_{i\in[N], i\neq i^*}$

```
1: Let $N = 2^D$
2: if $i^* \leq N/2$ then
3:     $(\texttt{seed}_i)_{i\in[N/2], i\neq i^*} = \texttt{PRGPartialTreeExpand}((\texttt{seed}_i)_{i\in[D-1]})$
4:     $(\texttt{seed}_i)_{i\in[N/2+1,N]} = \texttt{PRGTreeExpand}(\texttt{seed}_D, N/2, \mathsf{salt}, e)$
5: else
6:     $(\texttt{seed}_i)_{i\in[N/2]} = \texttt{PRGTreeExpand}(\texttt{seed}_1, N/2, \mathsf{salt}, e)$
7:     $(\texttt{seed}_i)_{i\in[N/2+1,N], i\neq i^*} = \texttt{PRGPartialTreeExpand}((\texttt{seed}_i)_{i\in[2,D]})$
8: return $(\texttt{seed}_i)_{i\in[N], i\neq i^*}$
```

4.9 Protocol specification

Algorithm 20 `MIRA_keygen`

```
Input: Security level $\lambda$
Output: Secret key sk, public key pk
 1: sk_seed $=$ RandomBytes($\lambda$)
 2: pk_seed $=$ RandomBytes($\lambda$)
 3: pk_PRG.Init(pk_seed)
 4: L_bytes $= \lceil \frac{k(mn-k)}{2} \rceil$
 5: $\boldsymbol{L'}$ = FqMatrixFromBytes(k, mn-k, pk_PRG.GetBytes(L_bytes))
 6: $\boldsymbol{L} = (I_k|\boldsymbol{L'})$
 7: for $i \in [k]$ do
 8:     for $j \in [m]$ do
 9:         for $\ell \in [n]$ do
10:             $(\boldsymbol{M}_i)_{j,\ell} = \boldsymbol{L}_{i,(j-1)n+\ell}$
11: $(\boldsymbol{E}, \_, \boldsymbol{y})$ = SampleSecretInstance(sk_seed)
12: $\boldsymbol{F} = \boldsymbol{E} - \sum_{i=1}^{k} \boldsymbol{y}_i \boldsymbol{M}_i$
13: for $i \in [k]$ do
14:     $f_i = \boldsymbol{F}_{1+\lfloor \frac{i-1}{n} \rfloor, 1+((i-1) \mod n)}$
15: $\boldsymbol{M}_0 = \boldsymbol{F} - \sum_{i=1}^{k} f_i \boldsymbol{M}_i$
16: pk = pk_seed || FqMatrixToBytes($\boldsymbol{M}_0$)
17: sk = sk_seed || pk_seed
18: return sk, pk
```

Algorithm 21 `MIRA_Sign`

Input: Secret key sk, public key pk, message $m \in \{0,1\}^*$
Output: Signature $\sigma \in \{0,1\}^*$

```
    ▷ Step 0: Parse keys
1:  $(\boldsymbol{M}_0, \ldots, \boldsymbol{M}_k, \boldsymbol{x}, \boldsymbol{\beta}, \boldsymbol{e}) = \texttt{ParseSecretKey}(\mathsf{sk})$

    ▷ Step 1: Commitment
2:  $\texttt{salt} = \texttt{RandomBytes}(2\lambda)$
3:  $\texttt{mseed} = \texttt{RandomBytes}(\lambda)$
4:  $\texttt{md} = \texttt{Hash}(\texttt{DS\_M}, m)$
5:  $(\texttt{seed}^{(e)})_{e\in[\tau]} = \texttt{SampleSeeds}(\lambda, \tau, \texttt{mseed})$
6:  for $e \in [\tau]$ do
7:      $(\texttt{seed}_i^{(e)})_{i\in[N]} = \texttt{PRGTreeExpand}(\texttt{seed}^{(e)}, N, \mathsf{salt}, e)$
8:      for $i \in [N-1]$ do                                    ▷ Compute leaf shares
9:          $(\boldsymbol{x}^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i]) = \texttt{SampleShares}(\mathsf{seed}_i^{(e)})$
10:         $\mathsf{cmt}_i^{(e)} = \texttt{Hash}(\texttt{DS\_C}, \mathsf{salt}, e, i, \mathsf{seed}_i^{(e)})$
11:     $\texttt{PRG.Init}(\texttt{seed}_N^{(e)})$
12:     $\texttt{abytes} = \texttt{PRG.GetBytes}(\lceil \frac{rm}{2} \rceil)$
13:     $\boldsymbol{a}^{(e)}[N] = \texttt{FqmArrayFromBytes(r, abytes)}$            ▷ Compute final leaf shares
14:     $\boldsymbol{a}^{(e)} = \sum_{i\in[N]} \boldsymbol{a}^{(e)}[i]$
15:     $\boldsymbol{x}^{(e)}[N] = \boldsymbol{x} - \sum_{i=1}^{N-1} \boldsymbol{x}^{(e)}[i]$
16:     $\boldsymbol{\beta}^{(e)}[N] = \boldsymbol{\beta} - \sum_{i=1}^{N-1} \boldsymbol{\beta}^{(e)}[i]$
17:     $c^{(e)}[N] = -\langle \boldsymbol{a}^{(e)}, \boldsymbol{\beta} \rangle - \sum_{i=1}^{N-1} c^{(e)}[i]$
18:     $\mathsf{cmt}_N^{(e)} = \texttt{Hash}(\texttt{DS\_C}, \mathsf{salt}, e, N, \mathsf{seed}_i^{(e)}, \boldsymbol{x}^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$
19:     $(\hat{\boldsymbol{x}}^{(e)}[\delta], \hat{\boldsymbol{\beta}}^{(e)}[\delta], \hat{\boldsymbol{a}}^{(e)}[\delta], \hat{c}^{(e)}[\delta])_{\delta\in[D]} = \texttt{Hypercube.MainShares-Compute}((\boldsymbol{x}^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i])_{i\in[N]})$
20: $h_1 = \texttt{Hash}(\texttt{md}, \mathsf{pk}, \mathsf{salt}, \texttt{DS\_1}, \mathsf{cmt}_1^{(1)}, \ldots, \mathsf{cmt}_N^{(1)}, \mathsf{cmt}_1^{(2)}, \ldots, \mathsf{cmt}_1^{(\tau)}, \cdots, \mathsf{cmt}_N^{(\tau)})$    ▷ Commit hypercube

    ▷ Step 2: First challenge
21: $(\boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})_{e\in[\tau]} = \texttt{SampleFirstChallenge}(h_1)$

    ▷ Step 3: First response
22: for $e \in [\tau]$ do
23:     $\boldsymbol{\alpha}^{(e)} = \texttt{MPC-MinRank.ComputeAlpha}(\boldsymbol{e}, \boldsymbol{a}^{(e)}, \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
24:     $(\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]} = \texttt{MPC-MinRank.Exec}\big[(\hat{\boldsymbol{x}}^{(e)}[\delta], \hat{\boldsymbol{\beta}}^{(e)}[\delta], \hat{\boldsymbol{a}}^{(e)}[\delta], \hat{c}^{(e)}[\delta])_{\delta\in[D]}, \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)}, \boldsymbol{\alpha}^{(e)}, \boldsymbol{M}_0, \ldots, \boldsymbol{M}_k\big]$
25: $h_2 = \texttt{Hash}(\texttt{md}, \mathsf{pk}, \mathsf{salt}, h_1, \texttt{DS\_2}, (\boldsymbol{\alpha}^{(e)}, (\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]})_{e\in[\tau]})$

    ▷ Step 4: Second challenge
26: $(i^{*(e)})_{e\in[\tau]} = \texttt{SampleSecondChallenge}(h_2)$

    ▷ Step 5: Second response
27: for $e \in [\tau]$ do
28:     $\texttt{ptree}^{(e)} = \texttt{PRGPartialTreeReveal}(\texttt{seed}^{(e)}, i^{*(e)})$
29:     $\boldsymbol{E}^{(e)}[i^{*(e)}] = \begin{cases} \boldsymbol{M}_0 + \sum_{j=1}^{k} \boldsymbol{x}^{(e)}[i^{*(e)}]\boldsymbol{M}_j & \text{if } i^{*(e)} = 1 \\ \sum_{j=1}^{k} \boldsymbol{x}^{(e)}[i^{*(e)}]\boldsymbol{M}_j & \text{otherwise} \end{cases}$
30:     $\boldsymbol{e}^{(e)}[i^{*(e)}] = \texttt{FqMatrixToFqmVector}(\boldsymbol{E}^{(e)}[i^{*(e)}])$
31:     $\texttt{MPC-MinRank.ComputeAlpha}(\boldsymbol{e}^{(e)}[i^{*(e)}], \boldsymbol{a}^{(e)}[i^{*(e)}], \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
32:     if $i^{*(e)} = N$ then
33:         $\mathsf{rsp}^{(e)} = (\texttt{ptree}^{(e)}, \mathsf{cmt}_{i^{*(e)}}^{(e)}, \boldsymbol{\alpha}^{(e)}[i^{*(e)}], \mathbf{0}, \mathbf{0}, 0)$
34:     else
35:         $\mathsf{rsp}^{(e)} = (\texttt{ptree}^{(e)}, \mathsf{cmt}_{i^{*(e)}}^{(e)}, \boldsymbol{\alpha}^{(e)}[i^{*(e)}], \boldsymbol{x}^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$

    ▷ Step 6: Signature
36: return $\sigma = (\texttt{salt}, h_1, h_2, (\mathsf{rsp}^{(e)})_{e\in[\tau]})$
```

Algorithm 22 `MIRA_Verify`

Input: Public key pk, message $m \in \{0,1\}^*$, signature $\sigma \in \{0,1\}^*$
Output: Accept or Reject

```
    ▷ Step 0: Parse keys
 1: $(\boldsymbol{M}_0, \cdots, \boldsymbol{M}_k) = \texttt{ParsePublicKey}(\mathsf{pk})$

    ▷ Step 1: Parse challenges
 2: $(\boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})_{e\in[\tau]} = \texttt{SampleFirstChallenge}(h_1)$
 3: $(i^{*(e)})_{e\in[\tau]} = \texttt{SampleSecondChallenge}(h_2)$
 4: for $e \in [\tau]$ do
 5:     if $i^{*(e)} = N$ then
 6:         Check that $(\boldsymbol{x}^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N]) = (\mathbf{0}, \mathbf{0}, 0)$. If not, return Reject.

    ▷ Step 2: Recompute $h_1$
 7: $\texttt{md} = \texttt{Hash}(\texttt{DS\_M}, m)$
 8: for $e \in [\tau]$ do
 9:     $(\mathsf{seed}_i^{(e)})_{i\in[N], i\neq i^{*(e)}} = \texttt{PRGPartialTreeExpand}(\mathsf{ptree}^{(e)}, N, \mathsf{salt}, e)$
10:     for $i \in [N], i \neq i^{*(e)}$ do
11:         if $i \neq N$ then
12:             $(\boldsymbol{x}^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i]) = \texttt{SampleShares}(\mathsf{seed}_i^{(e)})$
13:             $\mathsf{cmt}_i^{(e)} = \texttt{Hash}(\texttt{DS\_C}, \mathsf{salt}, e, i, \mathsf{seed}_i^{(e)})$
14:         else
15:             $\texttt{PRG.Init}(\mathsf{seed}_N^{(e)})$
16:             $\mathsf{abytes} = \texttt{PRG.GetBytes}(\lceil \frac{rm}{2} \rceil)$
17:             $\boldsymbol{a}^{(e)}[N] = \texttt{FqmArrayFromBytes(r, abytes)}$
18:             $\mathsf{cmt}_N^{(e)} = \texttt{Hash}(\texttt{DS\_C}, \mathsf{salt}, e, N, \mathsf{seed}_i^{(e)}, \boldsymbol{x}^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$
19: $\bar{h}_1 = \texttt{Hash}(\texttt{DS\_1}, \texttt{md}, \mathsf{pk}, \texttt{salt}, \mathsf{cmt}_1^{(1)}, \ldots, \mathsf{cmt}_N^{(1)}, \mathsf{cmt}_1^{(2)}, \ldots, \mathsf{cmt}_1^{(\tau)}, \cdots, \mathsf{cmt}_N^{(\tau)})$

    ▷ Step 3: Recompute $h_2$
20: for $e \in [\tau]$ do
21:     for $i \in [N], i \neq i^{*(e)}$ do
22:         $\boldsymbol{E}^{(e)}[i] = \begin{cases} \boldsymbol{M}_0 + \sum_{j=1}^{k} \boldsymbol{x}^{(e)}[i]\boldsymbol{M}_j & \text{if } i = 1 \\ \sum_{j=1}^{k} \boldsymbol{x}^{(e)}[i]\boldsymbol{M}_j & \text{otherwise} \end{cases}$
23:         $\boldsymbol{e}^{(e)}[i] = \texttt{FqMatrixToFqmVector}(\boldsymbol{E}^{(e)}[i])$
24:         $z^{(e)}[i] = -\sum_{j=1}^{n} \gamma_j^{(e)} \boldsymbol{e}_j^{(e)}[i]^{q^r}$
25:         $\boldsymbol{\alpha}^{(e)}[i] = \texttt{MPC-MinRank.ComputeAlpha}(\boldsymbol{e}^{(e)}[i], \boldsymbol{a}^{(e)}[i], \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
26:     $\boldsymbol{\alpha}^{(e)} = \sum_i \boldsymbol{\alpha}^{(e)}[i]$
27:     for $i \in [N], i \neq i^{*(e)}$ do
28:         $v^{(e)}[i] = \epsilon^{(e)} \cdot z^{(e)}[i] - \langle \boldsymbol{\alpha}^{(e)}, \boldsymbol{\beta}^{(e)}[i] \rangle - c^{(e)}[i]$
29:     $v^{(e)}[i^{*(e)}] = -\sum_{i\neq i^{*(e)}} v^{(e)}[i]$
30:     $(\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]} = \texttt{Hypercube.MainAlphaAndV-Compute}((\boldsymbol{\alpha}^{(e)}[i], v^{(e)}[i])_{i\in[N]})$
31: $\bar{h}_2 = \texttt{Hash}(\texttt{DS\_2}, \texttt{md}, \mathsf{pk}, \texttt{salt}, h_1, (\boldsymbol{\alpha}^{(e)}, (\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]})_{e\in[\tau]})$

    ▷ Step 4: Verify commitments
32: return $\bar{h}_1 \overset{?}{=} h_1 \wedge \bar{h}_2 \overset{?}{=} h_2$
```

5 Signature parameters

Our signature scheme uses the following parameters:

- the power of a prime number, $q \in \mathbb{N}$, to build $\mathbb{F}_q$;
- a positive integer, $m \in \mathbb{N}$, the number of rows of our matrices;
- a positive integer, $n \in \mathbb{N}$, the number of columns of our matrices;
- a positive integer, $k \in \mathbb{N}$, the length of the secret vector $\boldsymbol{x}$, and $k + 1$ is the number of matrices in the public key;
- a positive integer, $r \in \mathbb{N}$, the rank of the matrix $\boldsymbol{E}$;
- a positive integer, $N \in \mathbb{N}$, the number of parties simulated in the MPC protocol;
- a positive integer, $\eta \in \mathbb{N}$, the extension degree to build $\mathbb{F}_{q^{m \cdot \eta}}$;
- a positive integer, $\tau \in \mathbb{N}$, the number of rounds in the signature.

In order to choose the parameters, we need to consider:

- The security of the MinRank instance, i.e, the complexity of the attacks on the chosen parameters;
- The security of the signature scheme, i.e, the cost of a forgery;
- The size of the signature.

5.1 MIRA-Additive

We quickly remind here the choice of parameters, as it is already explained in [ABCD$^+$23]. $q = 16$ is the most efficient value, and m and n are chose without constraint. Then, r is chosen (5 to 6 here), and k is set as $k = (m-r)(n-r)-1$, i.e, the Rank Gilbert-Varshamov bound. Then, the security of the parameter set must be high enough.

Once the MinRank instance parameters are chosen, we set $N = 256$ for a short version of the signature, and $N = 32$ for a fast one. Then, τ and η need to be such that the security level of the signature is high enough (Section 9.1). We remind that the security levels of the signature 1,3, and 5, correspond respectively to the security of `AES-128`, `AES-192`, and `AES-256`. Finally, we settled on the following parameters:

Instance	**NIST Security Level**	q	m	n	k	r	N	η	τ	sk	pk	σ
`MIRA-128F`	1	16	16	16	120	5	32	1	28	16 B	84 B	7.376 B
`MIRA-128S`	1	16	16	16	120	5	256	1	18	16 B	84 B	5.640 B
`MIRA-192F`	3	16	19	19	168	6	32	1	41	24 B	121 B	15.540 B
`MIRA-192S`	3	16	19	19	168	6	256	1	26	24 B	121 B	11.779 B
`MIRA-256F`	5	16	23	22	271	6	32	1	54	32 B	150 B	27.678 B
`MIRA-256S`	5	16	23	22	271	6	256	1	34	32 B	150 B	20.762 B

Table 3: Parameters for MIRA with additive sharing, fast and short signatures

One should note that we obtain slightly larger signature size, due to rounding of bytes when m is odd. For `MIRA-192F`, we have 15.560 B, `MIRA-192S` gives 11.792 B, `MIRA-256F`, 27.732 B and `MIRA-256S` 20.796 B.
We also give some theoretical signature sizes for other parameters, such as the following one, used in [ARZV22] achieving the security level 1. With $(q, m, n, k, r) = (16, 16, 16, 142, 4)$, it is possible to obtain 5.550 Bytes (with short MIRA).
For the security level 3, it would also be possible to use the parameter $(16, 19, 19, 195, 5)$, which gives a size of 11.636 Bytes, however, the security of the instance is lower than the parameter we chose.
Overall, although it is possible to obtain slightly smaller signature sizes with other parameters, we choose to get slightly bigger signature (less than 100 bytes of difference) with a more conservative security (around 10 bits higher). We refer to the section 9.2, Table6 for the security of our parameter sets.

5.2 MIRA-Threshold

Note that in [ABCD$^+$23], two different variant of MIRA are described, using either an additive sharing scheme, or a threshold sharing scheme. The present specification contains only the additive sharing scheme, since so far, "threshold MPCitH" does not bring a significant advantage compared to "hypercube MPCitH" for the considered protocol for the MinRank problem: with the threshold technique, we obtain faster verification but with a degraded signature size, for similar signing time. This might however change in the future, in the case where threshold techniques were to improve.

6 Performance Analysis

This section provides performance measures of our implementations of MIRA.

Benchmark platform. Benchmark platform. The benchmarks have been performed on an Intel 13th Gen Intel (R) Core(TM) i9-13900K machine with 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 11.3.0) and uses the XKCP and OpenSSL (version 3.0.2) libraries.

For each parameter set, the results have been obtained by computing the mean from 10 random instances. To minimize biases from background tasks running on the benchmark platform, each instance has been repeated 10 times and averaged.

Constant time. The provided implementations have been implemented in a constant time way whenever relevant, and as such, the running time should not leak any information concerning sensitive data. For instance, all If branches depend on public data. Additionally, Valgrind (version 3.18.1) and LibVEX were used to check that there were no memory leaks on the implementation.

6.1 Reference Implementation

The performance concerning the reference implementation on the aforementioned benchmark platform are described in Table 4. The following optimization flags have been used during compilation:

– Concerning the C code: `-O3 -flto`.

– Concerning the ASM code (required in the XKCP library): `-x assembler-with-cpp -Wa,-defsym,old_gas_syntax=1 -Wa,-defsym,no_plt=1`.

Instance	Key Generation	Sign	Verify
`MIRA-128F`	104.5 K	38.9 M	38.0 M
`MIRA-128S`	104.3 K	53.9 M	50.0 M
`MIRA-192F`	281.6 K	134.3 M	134.1 M
`MIRA-192S`	280.3 K	154.6 M	152.9 M
`MIRA-256F`	675.6 K	342.4 M	350.0 M
`MIRA-256S`	665.7 K	387.6 M	376.4 M

Table 4: Thousand (K) and Million (M) of CPU cycles of MIRA reference implementation.

6.2 Optimized Implementation

The performance concerning the reference implementation on the aforementioned benchmark platform are described in Table 5. The following optimization flags have been used during compilation:

– Concerning the C code: `-O3 -flto -mavx2 -mpclmul -msse4.2 -maes`.

– Concerning the ASM code (required in the XKCP library): `-x assembler-with-cpp -Wa,-defsym,old_gas_syntax=1 -Wa,-defsym,no_plt=1`.

Instance	Key Generation	Sign	Verify
MIRA-128F	112.0 K	37.4 M	36.7 M
MIRA-128S	112.0 K	46.8 M	43.9 M
MIRA-192F	288.8 K	107.2 M	107.0 M
MIRA-192S	286.3 K	119.7 M	116.2 M
MIRA-256F	706.0 K	322.3 M	323.2 M
MIRA-256S	694.8 K	337.7 M	331.4 M

Table 5: Thousand (K) and Million (M) of CPU cycles of MIRA optimized implementation.

7 Known Answer Test Values

Known Answer Test (KAT) values have been generated using the script provided by the NIST and can be retrieved in the KATs/ folder. Both reference and optimized implementations generate the same KATs. In addition, examples with intermediate values have also been provided in these folders. The intermediate values correspond with one execution calling the NIST-provided randombytes function seeded with zero.

Notice that one can generate the test files as mentioned above using the kat and verbose modes of the implementation, respectively. The procedure is detailed in the technical documentation (README file of the provided code).

8 Expected security strength

Our scheme relies on the hardness of solving a MinRank instance. We expect our parameters to offer the security required for each security category. For the hardness of solving a MinRank instance, we refer to the following section, as well as the design document [ABCD$^+$23], where we describe the best attacks on the MinRank problem, namely, the kernel attack, the Support Minor Modeling, and the Minors Modeling.
For the hardness of forging the signature, we refer to the following section and the design document, which gives the complexity of the attack on the Fiat-Shamir transform described in [KZ20]. This is the best attack on the Fiat-Shamir signature, and as such, this allows us to choose the parameters τ and η accordingly.
Moreover, our signature scheme originates from a zero-knowledge proof of knowledge. The proofs of soundness and zero-knowledge of the proof of knowledge can be found in [ABCD$^+$23].
For the unforgeability of the scheme, we have the following theorem (the hash functions H_i correspond, in order, to the hash function we use in the scheme):

Theorem 1. *Let the PRG used be (t, ϵ_{PRG})-secure, and assume an adversary running in time t has advantage at most ϵ_{MR} against the underlying MinRank problem. Let further assume that $\mathsf{H}_0, \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3, \mathsf{H}_4$ behave as random oracles, with an output of 2λ bits. Then,* *if an adversary makes q_i queries to H_i, q_S queries to the signing oracle, and runs in time at most t, their probability to produce a forgery (EUF-CMA) for the MIRA Additive Signature Scheme is upper bounded as:*

$$\mathsf{Pr}[\mathsf{Forge}] \leq \frac{3 \cdot (q + \tau \cdot N \cdot q_S)^2}{2 \cdot 2^{2\lambda}} + \frac{q_S \cdot (q_S + 5q)}{2^{2\lambda}} + q_S \cdot \tau \cdot \epsilon_{PRG} + \mathsf{Pr}[X + Y = \tau] + \epsilon_{MR}$$

with τ the number of rounds of the signature, $X = \max_{i \in [0,q_2]}\{X_i\}$ with $X_i \sim \mathcal{B}(\tau, p)$, and $Y = \max_{i \in [0,q_4]}\{Y_i\}$ with $Y_i \sim \mathcal{B}(\tau - X, \frac{1}{N})$.

We refer to the design document ([ABCD$^+$23]) for a proof of this theorem.
To resume, we have an EUF-CMA secure signature scheme, relying on the hardness of solving the MinRank scheme. For each security level (1,3, and 5), the parameters are taken such that:

- Solving the MinRank instance comply to the target NIST security category;
- Forging a valid signature without knowing the secret key (with high probability) requires 2^λ hash computations under the best known forgery attack;
- Access to a signing oracle does not help an attacker to forge a signature.

Our signature scheme relies on the security of hash functions and commitments functions as well. In the unforgeability security proof, they are modelled as random oracles. In practice, we use `SHAKE` and `SHA3` functions, which are collision resistant and preimage resistant. With Grover's algorithm, the complexity to find a preimage is of $\mathcal{O}(2^{\frac{n}{2}})$ (for an hash function with an n bits output), while collisions can be found in $\mathcal{O}(2^{\frac{n}{3}})$ with Brassard's algorithm [BHT98]. However, this last algorithm is unlikely to perform better than $\mathcal{O}(2^{\frac{n}{2}})$ in practice (see [Ber09]). Hence, as many other cryptosystems, we consider here that a hash function with an output of $2 \cdot \lambda$ bits provides a quantum security of λ bits.

8.1 Resistance to quantum attacks

There exist two type of attacks for our problem: combinatorial and algebraic attacks.

Since combinatorial attacks rely for their exponential part on guessing special vector spaces of a whole space, they can be improved by a square root factor through the Grover algorithm, which basically means dividing by a factor 2 the exponential part of the complexity. For algebraic attacks such an improvement cannot be obtained directly. Notice that, for the type of parameters we consider which are close to the Gilbert-Varshamov bound, combinatorial and algebraic attack behave similarly. Therefore, in practice, resistance to quantum attacks shall be obtained by roughly dividing by 2 the classical security level.

9 Analysis of known attacks

9.1 Kales and Zaverucha attack on the Fiat-Shamir transform

There are several attacks against signatures from zero-knowledge proofs obtained thanks to the Fiat-Shamir heuristic. [AABN02] propose an attack more efficient than the bruteforce one for protocols with more than one challenge, i.e. for protocols of a minimum of 5 rounds. However, it is not the most efficient.

Kales and Zaverucha proposed in [KZ20] a forgery achieved by guessing separately the two challenge of the protocol. It results 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 tradeoff 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 challenge τ' times among the τ repetitions and the second challenge one time.

In our case, one obtains the following cost of forgery:

$$\text{cost}_{\text{forge}} = \min_{0\leq\tau_1\leq\tau} \left\{ \frac{1}{\sum_{i=\tau_1}^{\tau} \binom{\tau}{i} p^i (1-p)^{\tau-i}} + (N)^{\tau-\tau_1} \right\}$$

where p is the false positive rate of the protocol Π^η, i.e, $p = \frac{2}{q^{m\eta}} - \frac{1}{q^{2m\eta}}$

9.2 Attacks on MinRank

We describe in this section the most effective attacks on MinRank. To begin with, there is an approach, which is efficient on all the attacks, namely, the hybrid approach.

Hybrid approach As shown in [BBB+22, Section 5], solving a MinRank problem of parameters (q, m, n, K, r) amount to solve q^{ar} smaller MinRank problems of parameters $(q, m, n-a, K-am, r)$. The idea is to multiply the matrix $\boldsymbol{M}$ on the right by a random $n \times n$ invertible matrix $\boldsymbol{P}$ and make the bet that the new matrix has its first a columns equal to zero:

$$MP = \begin{pmatrix} 0_{m\times a} & \tilde{M} \end{pmatrix}.$$

As $\boldsymbol{M}$ has rank r, it can be written $\boldsymbol{M} = \boldsymbol{SC}$ with $\boldsymbol{S} \in \mathbb{F}_q^{m\times r}$ of rank r and $\boldsymbol{C} \in \mathbb{F}_q^{r\times n}$, and cancelling the first a columns of $\boldsymbol{MP}$ corresponds to the cancellation of the first a columns of $\boldsymbol{CP}$, which has a probability $\Omega(q^{-ar})$ to happen. This leads to a smaller MinRank problem, obtained by eliminating am variables x_i (using the am linear equations coming from the cancellation of the a columns of $\boldsymbol{MP}$) in the matrix $\tilde{\boldsymbol{M}}$, that has still rank r. This new problem has parameters $(q, m, n-a, K-am, r)$.

If a solution to the small instance is found, then the solution of the original problem can be recovered by mutiplying it on the right by $\boldsymbol{P}^{-1}$. The cost of this approach is

$$\min_a \left(q^{ar}\mathbb{C}_{\mathcal{A}}(q, m, n-a, K-am, r)\right) \tag{1}$$

where $\mathbb{C}_{\mathcal{A}}(q, m, n, K, r)$ is the cost of an algorithm $\mathcal{A}$ to solve a MinRank problem.

The cost of the transformation is $2mn^{\omega-1} + 3n^{\omega}$ (for the multiplication by P, computation and multiplication by $\boldsymbol{P}^{-1}$) $+3(am)^{\omega-1}K$ (for the linearization of the am linear equations) $+am^2(n-a)(K-am)$ (for the elimination of am variables x), which is negligeable.

9.2.1 The kernel attack

The kernel attack was described by Goubin and Courtois in [GC00]. The idea of the attack is to take random vectors, and hoping that they are in the kernel of $\boldsymbol{E}$. Since $\boldsymbol{E}$ is of size $m \times n$, and is of rank at most r, $\mathrm{Ker}(\boldsymbol{E})$ will be a matrix of dimensions $n \times (n-r)$ at least. Thus, if $v \xleftarrow{\$} \mathbb{F}_q^n$, $\mathsf{Pr}[v \in \mathrm{Ker}(\boldsymbol{E})] = \frac{q^{n-r}}{q^n} = \frac{1}{q^r}$. Then, if we get l independant vectors in $\mathrm{Ker}(\boldsymbol{E})$, and set the matrix $\boldsymbol{X}$ whose columns are the l vectors, we can compute $(\boldsymbol{M}_0 + \sum_{i=1}^k x_i \boldsymbol{M}_i)X$, which gives us a linear system in $x_1 \dots x_k$, and $m \cdot l$ equations (since we have $(\boldsymbol{M}_0 + \sum_{i=1}^k x_i \boldsymbol{M}_i)X = 0$). Thus with $l = \lceil \frac{k}{m} \rceil$, we have a unique solution to the system, which we can find with linear algebra.
As to the complexity of the attack, it is quite obvious that it is in $O(q^{r\lceil \frac{k}{m} \rceil})$ to find the vectors in the kernel, and in $O(k^{\omega})$ to solve the linear system. Hence, the total complexity is

$$O(q^{r\lceil \frac{k}{m} \rceil} k^{\omega})$$

9.2.2 Algebraic attacks

Kipnis-Shamir Modeling The first algebraic modeling was proposed in [KS99]. It consists in solving with algebraic techniques the system coming from the kernel attack, the unknowns being the x_i's and the entries of the matrix $\boldsymbol{X}$. However, its complexity is not well understood, in particular because, according to [BB22], 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 [FSS10, FSS13]. The algebraic system consists of all the minors of size $r+1$ of the formal matrix $\boldsymbol{M} = \boldsymbol{M}_0 + \sum_{i=1}^k x_i \boldsymbol{M}_i$, whose entries are linear forms in the unknowns x_i's. This is a determinantal system, whose Hilbert series is given by

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

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

As long as $k < (m-r)(n-r)$, the series is a polynomial, and the degree of regularity D of the system is $\deg(HS)+1$. The complexity of the Gröbner basis computation can then be estimated as the cost of computing the echelon form on the Macaulay matrix of the system in degree D that has $\binom{k+D}{D}$ columns and almost the same rank. As shown in [GND23], it is possible to refine the F5 algorithm to construct subs-matrices of the Macaulay matrices with a number of rows equal to its rank. Even if this algorithm has not been designed yet for the Minors modeling for any parameters set, we estimate the complexity of computing the echelon form as

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

with ω the linear algebra constant. In all our estimates, we use conservatively $\omega = 2$.

Support Minors Modeling The Support Minors modeling was introduced in [BBC$^+$20]. The idea is to consider the vector space generated by the rows of $\boldsymbol{M}$, that has rank r, to introduce a formal matrix $\boldsymbol{C} \in \mathbb{F}_q[c_{i,j}]^{r\times n}$ representing a generator matrix of this vector space. As $\boldsymbol{C}$ is a basis for the rows of $\boldsymbol{M}$, each row m_i of $\boldsymbol{M}$ is a linear combination of the rows of $\boldsymbol{C}$. This leads to the algebraic system

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

Considering the maximal minors of $\boldsymbol{C}$ as new variables, the system consists of bilinear equations in the x_i's and the minors of $\boldsymbol{C}$. It is conjectured in [BBC$^+$20], 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 a sub-matrix of the Macaulay matrix at augmented degree $b \le \min(q-1, r+1)$ in the x_i's, that contains N_b rows and M_b columns:

$$N_b = \sum_{i=1}^{b} (-1)^{i+1} \binom{n}{r+i} \binom{k+b-1-i}{b-i} \binom{m+i-1}{i}. \tag{2}$$

$$M_b = \binom{k+b-1}{b} \binom{n}{r}. \tag{3}$$

The degree b is computed as the smallest degree such that $N_b \ge M_b - 1$. In this case, the final cost in $\mathbb{F}_q$ operations is given by

$$\mathcal{O}\left(N_b {M_b}^{\omega-1}\right),$$

where ω is the linear algebra constant.

Instance	NIST Security Level	q	m	n	k	r	a	kernel	a	b	SM	a	D	Minors
`MIRA-128`	1	16	16	16	120	5	8	167	7	1	169	4	9	**164**
`MIRA-192`	3	16	19	19	168	6	9	225	6	7	231	6	9	**221**
`MIRA-256`	5	16	23	22	271	6	12	297	10	4	302	7	13	**290**

Table 6: Complexity of the MinRank attacks for the proposed parameters. a is the hybrid parameter used for each attack, b the degree for the Support Minors modeling, D the degree for the Minors modeling. The complexities are given in $\log_2$ bit operations. We use $\omega = 2$.

We observe in Table 6 that in the range of parameters around the Rank Gibert-Varshamov bound ($k = (n-r)(m-r)-1$), all attacks give the same order of complexity, the Minors modeling being a little more efficient as it uses only one set of variables.

10 Advantages and limitations

10.1 Advantages

The MIRA scheme enjoys the following advantages:

Difficulty of the problem. MinRank is an NP-Complete problem that has been significantly studied for some time. It already has a central role in cryptography, being used in several cryptosystems, and in some attacks in multivariate cryptography. The MIRA scheme relies on the hardness random non-structured MinRank instance which is a conservative assumption.

No cyclic structure. In particular, and unlike other cryptosystems, the MinRank instance used in MIRA does not rely on a cyclic structure for which the quantum security is more questionable.

Size of public keys + signature. The sum of the sizes of public keys and signature is, for example, less than twice as big as in Dilithium for the NIST security level 1, thanks to our small public key (3.7 kB for Dilithium2 compared to 5.7 kB for our level 1 security instance). As a result, it is one of the shortest signature among the MPCitH-based schemes.

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 mitigated impact on the total size of the signature. For example, the short signature size's is 5.6 kB for the set of parameters $(q, m, n, k, r) = (16, 16, 16, 120, 5)$ for the level 1 of security. If we choose the parameters $(q, m, n, k, r) = (16, 19, 19, 168, 6)$, which reaches more than 192 bits of security for the underlying cryptographic problem, the signature size will only be 6.3 kB to reach the level 1 of security. Thus, if one later discovers effective attacks against the MinRank problem that force us to increase the parameters, the size of the signatures will only be impacted that much.

10.2 Limitations

MIRA also suffers the following limitations:

Growth rate of the signature size. The size of the signature grows with a quadratic rate 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 be secure (we see that going from level 1 to 3, the number of repetitions is increased by almost half, while the MinRank instance grows as well).

Signature speed. Our scheme is slower than lattice-based signature schemes but compares well in general with other signature schemes.

Implementation. This scheme is relatively complex to implement. It might be particularly heavy for low-cost devices such as smart cards or embedded systems, although it has potential to perform well on hardware as being highly parallelizable.

References

AABN02. Michel Abdalla, Jee Hea An, Mihir Bellare, and Chanathip Namprempre. From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security. Cryptology ePrint Archive, Paper 2002/022, 2002. `https://eprint.iacr.org/2002/022`.

ABCD$^+$23. Nicolas Aragon, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, and Matthieu Rivain. MIRA : a Digital Signature Scheme based on the MinRank problem and the MPC-in-the-Head paradigm. arXiv, 2023.

AMGH$^+$22. Carlos Aguilar-Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The Return of the SDitH. Cryptology ePrint Archive, Paper 2022/1645, 2022. `https://eprint.iacr.org/2022/1645`.

ARZV22. Gora Adj, Luis Rivera-Zamarripa, and Javier Verbel. MinRank in the Head: Short Signatures from Zero-Knowledge Proofs. Cryptology ePrint Archive, Paper 2022/1501, 2022. `https://eprint.iacr.org/2022/1501`.

BB22. Magali Bardet and Manon Bertin. Improvement of Algebraic Attacks for Solving Superdetermined MinRank Instances. In Jung Hee Cheon and Thomas Johansson, editors, *pqcrypto 2022*, volume 13512 of *lncs*, pages 107–123, Cham, September 2022. Springer International Publishing.

BBB$^+$22. Magali Bardet, Pierre Briaud, Maxime Bros, Philippe Gaborit, and Jean-Pierre Tillich. Revisiting Algebraic Attacks on MinRank and on the Rank Decoding Problem. Cryptology ePrint Archive, Paper 2022/1031, 2022. `https://eprint.iacr.org/2022/1031`.

BBC$^+$20. Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier Verbel. Improvements of Algebraic Attacks for Solving the Rank Decoding and MinRank Problems. In *Advances in Cryptology – ASIACRYPT 2020*, pages 507–536. Springer International Publishing, 2020.

Ber09. Daniel Bernstein. Cost analysis of hash collisions: Will quantum computers make sharcs obsolete. 01 2009.

BHT98. Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum Cryptanalysis of Hash and Claw-Free Functions. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*, volume 1380 of *Lecture Notes in Computer Science*, pages 163–169. Springer, 1998.

Fen22. Thibauld Feneuil. Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP. Cryptology ePrint Archive, Paper 2022/1512, 2022. `https://eprint.iacr.org/2022/1512`.

FR22. Thibauld Feneuil and Matthieu Rivain. Threshold Linear Secret Sharing to the Rescue of MPC-in-the-Head. Cryptology ePrint Archive, Paper 2022/1407, 2022. `https://eprint.iacr.org/2022/1407`.

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

FSS10. Jean-Charles Faugère, Mohab Safey El Din, and Pierre-Jean Spaenlehauer. Computing loci of rank defects of linear matrices using Gröbner bases and applications to cryptology. In *International Symposium on Symbolic and Algebraic Computation, ISSAC 2010, Munich, Germany, July 25-28, 2010*, pages 257–264, 2010.

FSS13. Jean-Charles Faugère, Mohab Safey El Din, and Pierre-Jean Spaenlehauer. On the complexity of the generalized minrank problem. *JSC*, 55:30–58, 2013.

GC00. Louis Goubin and Nicolas T. Courtois. Cryptanalysis of the TTM Cryptosystem. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2000.

GND23. Sriram Gopalakrishnan, Vincent Neiger, and Mohab Safey El Din. Refined f_5 algorithms for ideals of minors of square matrices, 2023.

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

KS99. Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In *crypto '99*, volume 1666 of *LNCS*, pages 19–30, Santa Barbara, California, USA, August 1999. Springer.

KZ20. Daniel Kales and Greg Zaverucha. An Attack on Some Signature Schemes Constructed From Five-Pass Identification Schemes. Cryptology ePrint Archive, Paper 2020/837, 2020. `https://eprint.iacr.org/2020/837`.

Loi07. Pierre Loidreau. Rank metric and cryptography. *HAL*, 2007, 2007.