RYDE specifications

Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Matthieu Rivain, Jean-Pierre Tillich, et al.

RYDE 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], Antoine Joux[8], Matthieu Rivain[7], Jean-Pierre Tillich[3], and Adrien Vinçotte[5]

[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
[8] CISPA, Helmholtz Center for Information Security, Saarbrücken

Table of Contents

1 Introduction

This document specifies the RYDE digital signature scheme. It is a quantum-resistant signature, based on zero-knowledge proofs, and symmetric functions, such as hash functions. We present a high-level description of the scheme and the MPC-in-the-Head paradigm (it allows to transform MPC protocols in proofs of knowledge) on which the signatures are based. The security of the signature relies on the Rank Syndrome Decoding problem: thanks to the previous process we can transform a proof of knowledge of a witness to a Rank-SD instance [Fen22] to obtain a signature scheme. We apply then the Fiat-Shamir transform to obtain signature schemes.

We present a signature scheme, of which we have designed two variants, both based on the transformation of an additive MPC protocol optimized with the hypercube technique [AMGH$^+$22]. The difference of these variants is the number of parties we use in the underlying MPC protocol: taking less parties increases the size of the signature, but induces a faster one. We refer to the design document [BCDF$^+$23] for the proofs of the security statements.

2 Background and notations

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

Let $\mathbb{F}_{q^m}$ be the finite field of size q^m. We fix a $\mathbb{F}_q$-basis $\mathcal{B} = (b_1, ..., b_m)$ of $\mathbb{F}_{q^m}$.

Let us consider a vector $\boldsymbol{x} = (x_1, ..., x_n) \in \mathbb{F}_{q^m}^n$. Each coordinate x_i can be associated with a vector $(x_{i,1}, ..., x_{i,m}) \in \mathbb{F}_q^m$ such that

$$x_i = \sum_{j=1}^{m} x_{i,j} b_j \ .$$

The matrix $\boldsymbol{M}(\boldsymbol{x}) = (x_{i,j})_{(i,j)\in[1,n]\times[1,m]}$ is called matrix associated to the vector $\boldsymbol{x}$.

We then define the following notions:

- the rank weight $W_R(\boldsymbol{x})$ of a vector $\boldsymbol{x}$ is defined as the rank of the associated matrix: $W_R(\boldsymbol{x}) := \mathsf{Rank}(\boldsymbol{M}(\boldsymbol{x}))$.
- the distance between two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is then given by $d(\boldsymbol{x}, \boldsymbol{y}) := W_R(\boldsymbol{x} - \boldsymbol{y})$.
- the support of a vector $\boldsymbol{x}$ is the linear subspace of $\mathbb{F}_{q^m}$ generated by its coordinates:

$$\mathsf{Supp}(\boldsymbol{x}) = \langle x_1, ..., x_n \rangle.$$

Now that we have defined the mathematical background, we can define the Rank Syndrome Decoding problem, on which the security of the signature RYDE is based:

Rank Syndrome Decoding problem:
Let (n, k, r) positive integers such that $k \leq n$. The Rank Syndrome Decoding (Rank-SD) problem with parameters (q, m, n, k, r) is the following one:

Let $\boldsymbol{H}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ such that:

- $\boldsymbol{H'}$ is uniformly sampled from $\mathbb{F}_{q^m}^{(n-k)\times k}$, and $\boldsymbol{H} = (I \mid \boldsymbol{H'}) \in \mathbb{F}_{q^m}^{(n-k)\times n}$
- $\boldsymbol{x}$ is uniformly sampled from $\{\boldsymbol{x} \in \mathbb{F}_{q^m}^n, W_R(\boldsymbol{x}) \leq r\}$
- $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$

From the public data $(\boldsymbol{H}, \boldsymbol{y})$, find $\boldsymbol{x}$.

In the RYDE signature scheme, the public key is an instance $(\boldsymbol{H}, \boldsymbol{y})$ of the Rank-SD problem, and the secret key is the corresponding solution $\boldsymbol{x}$. The principle of the signature is based on a transformation of a proof of knowledge into a non-interactive protocol thanks to the Fiat-Shamir transform [FS86]. Iterating this process multiple times gives the verifier high assurance that the prover knows $\boldsymbol{x}$.

The interactive proof relies on a prover which simulates a multiparty computation (MPC) protocol, where each party has a share of the secret witness $\boldsymbol{x}$. One details here the formalism proposed by [FR22]. The sharing of a secret value s into N parties is denoted $(s[1], ..., s[N])$, where $s[i]$ is the share of index i for $i \in [1, N]$, and $s[J] = (s[i])_{i \in J}$ is the subset of shares for $J \subset [1, N]$.

Let $\mathbb{F}$ a finite field and an integer $\ell \in [1, N]$. A (ℓ, N)-threshold Linear Secret Sharing Scheme is a method to share a secret $s \in \mathbb{F}$ into N shares $s[1, N] = (s[1], ..., s[N]) \in \mathbb{F}^N$ such that the secret can be rebuilt from any subset of $\ell + 1$ shares, while the knowledge of any subset of ℓ shares (or less) gives no information on s.

Formally, a $(\ell + 1, N)-$threshold LSSS is a pair of algorithms:

$$\begin{cases} \mathsf{Share} : \mathbb{F} \times R \to \mathbb{F}^N \\ \mathsf{Reconstruct}_J : \mathbb{F}^\ell \to \mathbb{F} \end{cases}$$

where R denotes some randomness space, and $\mathsf{Reconstruct}_J$ is indexed by a subset $J \subset [1, N]$ of $\ell + 1$ elements. These algorithms must verify the following properties:

- **Correctness**: $\mathsf{Reconstruct}$ allows to retrieve s if it has any subset of $\ell + 1$ shares as input;
- **Perfect ℓ-privacy**: The distribution of the outputs of Share on two different secrets are perfectly indistinguishable;
- **Linearity**: for every $v, v' \in \mathbb{F}^l$, $\alpha \in \mathbb{F}$ and subset $J \subset [1, N]$ of $\ell + 1$ elements:

$$\mathsf{Reconstruct}_J(\alpha v + v') = \alpha \mathsf{Reconstruct}_J(v) + \mathsf{Reconstruct}_J(v')$$

We present below the additive secret sharing, that we use in the signature scheme:

Additive secret sharing:
An additive secret sharing over a finite field $\mathbb{F}$ is an (N, N)-threshold LSSS where $R =$

$\mathbb{F}^{N-1}$. The Share algorithm is defined as:

$$\mathsf{Share} : (s, (r_1, ..., r_{N-1})) \to s[1, N] = \left(r_1, ..., r_{N-1}, s - \sum_{i=1}^{N-1} r_i \right)$$

The $\mathsf{Reconstruct}_{[1,N]}$ outputs the sum of all the shares.

3 High level description of the signature scheme

As mentioned below, the security of the signature scheme relies on the hardness to solve the Rank-SD problem. The instance $(\boldsymbol{H}, \boldsymbol{y})$ is the public key of the signature, and the associated witness $\boldsymbol{x}$ is the secret key. In order to sign a message, we must:

- Build an interactive zero-knowledge proof of the knowledge of $\boldsymbol{x}$;
- Use the Fiat-Shamir transform to obtain a non-interactive proof

The proof relies on an MPC protocol, and we use the MPC-in-the-Head paradigm to transform it in a zero-knowledge proof.

3.1 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 $x[\cdot]$ they received at the begin of the protocol, so that they obtained the shares $f(x)[\cdot]$ at the end of the protocol. 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 a hint which depends on the witness ω and the previous elements sent;
- Computing: since the sharing is linear, the parties can perform linear transformations on their shares;
- Broadcasting: it is sometimes necessary to open some shared values (which give no information about the witness ω) to execute the protocol.

Due to application to zero-knowledge proofs that we will do below, we suppose here that all parties are honest: they compute and share only correct values.

The interactive proof relies on the application of the the MPC-in-the-Head (MPCitH) paradigm [IKOS07]: the prover simulates a MPC protocol, where each party has a share of the secret witness $\boldsymbol{x}$. Informally, an MPC protocol takes place as follows:

- each party takes a share of the witness;
- they jointly compute (some operations are locally calculated, others are performed together) a Boolean value indicating whether $\boldsymbol{x}$ is right the witness associated to $(\boldsymbol{H}, \boldsymbol{y})$ (in other words, the MPC protocol compute the function f which verifies that $\boldsymbol{x}$ is the correct solution of the instance $(\boldsymbol{H}, \boldsymbol{y})$);

– In the case of the additive sharing, the view of all parties except one gives no information about $\boldsymbol{x}$.

We formally describe the MPC protocol [Fen22] used for the RYDE scheme in Figure 1. Compared to its original version, the protocol of Figure 1 integrates an optimization: it relies on the fact that $L(1) = 0$ to save one field element in the communication cost. We refer to [BCDF+23] for the proof of correctness and the false positive rate.

Public values. An instance of a Rank-SD problem: $\boldsymbol{H} = (I\ \boldsymbol{H'}) \in \mathbb{F}_{q^m}^{(n-k)\times n}$ and $\boldsymbol{y} \in \mathbb{F}_{q^m}^{n-k}$.

Inputs. Each party takes a share of $\boldsymbol{x}_B[\cdot]$ where $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$. Let U a $\mathbb{F}_q$-linear subspace of $\mathbb{F}_{q^m}$ of dimension r which contains $x_1, \ldots, x_n$ and has for annihilator polynomial $L = \prod_{u\in U}(X-u)$. Each party takes a share of the following sharing as inputs : $L[\cdot] = \sum_{i=1}^{r-1} \beta_i[\cdot](X^{q^i} - X) + (X^{q^r} - X)$, $\boldsymbol{a}[\cdot]$ where $\boldsymbol{a}$ is uniformly sampled from $\mathbb{F}_{q^{m\cdot\eta}}^r$ and $c[\cdot]$ such that $c = -\langle \boldsymbol{\beta}, \boldsymbol{a}\rangle$.

MPC Protocol:

1. The parties sample uniformly random: $(\gamma_1, ..., \gamma_n, \varepsilon) \xleftarrow{\$} \mathbb{F}_{q^{m\cdot\eta}}^{n+1}$.
2. The parties locally compute $\boldsymbol{x}_A[\cdot] = \boldsymbol{y} - \boldsymbol{H'}\boldsymbol{x}_B[\cdot]$ and $\boldsymbol{x}[\cdot] = (\boldsymbol{x}_A[\cdot] \,||\, \boldsymbol{x}_B[\cdot])$.
3. The parties locally compute $z[\cdot] = -\sum_{j=1}^n \gamma_j(x_j[\cdot]^{q^r} - x_j[\cdot])$.
4. The parties locally compute for all k from 1 to $r-1$: $\omega_k[\cdot] = \sum_{j=1}^n \gamma_j(x_j[\cdot]^{q^k} - x_j[\cdot])$.
5. The parties locally compute $\boldsymbol{\alpha}[\cdot] = \varepsilon \cdot \boldsymbol{\omega}[\cdot] + \boldsymbol{a}[\cdot]$.
6. The parties open $\boldsymbol{\alpha}$.
7. The parties locally compute $v[\cdot] = \varepsilon \cdot z[\cdot] - \langle \boldsymbol{\alpha}, \boldsymbol{\beta}[\cdot]\rangle - c[\cdot]$.
8. The parties open v.
9. The parties output ACCEPT if $v = 0$, and reject otherwise.

Fig. 1: Protocol Π^η: a protocol that checks that the given input is a solution of the considered instance of Rank-SD problem

3.2 Application of the MPCitH paradigm

The MPC-in-the-Head paradigm, introduced by [IKOS07], is a framework enabling to convert an MPC protocol into a zero-knowledge proof. The RYDE signature scheme has two trade-offs, induced by the same proof of knowledge. Both use an additive MPC protocol turned into zero-knowledge proof thanks to the MPC-in-the-Head paradigm, but with different numbers of parties.

We start by presenting the main idea of the hypercube optimization [AMGH+22]: splitting a secret in $N = 2^D$ additive shares, and arranging them in a geometrical structure. Let us consider an hypercube of dimension D, each dimension having 2 slots. These 2^D shares are called "leafs". Since we use an additive secret sharing, one can intertwine several executions where each party (what we will call a "main party") can be the sum of a subset of leafs, such that these subsets form a partition of the leafs. The main parties' shares of a secret s are denoted $\hat{s}$. We build here D intertwined protocols, and each of them has 2 main parties (an interested reader can find more details in [BCDF+23]).

The proof of knowledge in the additive case works as follows:

- the prover builds the inputs of the MPC protocol, i.e, he generates the states of N parties;
- he then commits the states of the parties, and receives a first challenge corresponding to some random values. This challenge consists in the random values $(\gamma_1, ..., \gamma_n, \varepsilon)$ to be used in the MPC protocol;
- the prover runs one MPC protocol per dimension of hypercube. For each of the D dimensions, he builds two main parties, each of them being the sum of the leafs whose has the same coordinate for this dimension. The prover hashes the shares of $\boldsymbol{\alpha}$ and $\boldsymbol{v}$;
- thanks to this challenge, he simulates the MPC protocol on the main parties, and hashes 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 specific share of data for the party whose views are not revealed;
- the verifier can verify that the computation of the MPC protocol was done honestly.

From the interactive proof of knowledge, we can easily build a non-interactive proof/signature, using the Fiat-Shamir transform [FS86].

4 Detailed algorithmic description

This section is dedicated to a low-level algorithmic description of the three algorithms of our scheme: `RYDE_Keygen`, `RYDE_Sign` and `RYDE_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:		
$\texttt{seed\_pk}$	$\{0,1\}^\lambda$	Seed for generation of the public matrix $\boldsymbol{H}$.
$\texttt{seed\_sk}$	$\{0,1\}^\lambda$	Seed for generation of the secret vector $\boldsymbol{x}$ and secret annihilator polynomial $\boldsymbol{\beta}$.
$\texttt{mseed}$	$\{0,1\}^\lambda$	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:		
$\texttt{DS\_M}$	$\{0,1\}^\lambda$	Domain separator for the message.
$\texttt{DS\_T}$	$\{0,1\}^\lambda$	Domain separator for the PRG tree.
$\texttt{DS\_C}$	$\{0,1\}^\lambda$	Domain separator for the PRG commitments.
$\texttt{DS\_1}$	$\{0,1\}^\lambda$	Domain separator for the first response.
$\texttt{DS\_2}$	$\{0,1\}^\lambda$	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:		
$\boldsymbol{H}$	$\mathbb{F}_{q^m}^{(n-k)\times n}$	Public matrix.
$\boldsymbol{x}$	$\mathbb{F}_{q^m}^{n}$	Secret vector of small rank weight.
$\boldsymbol{\beta}$	$\mathbb{F}_{q^m}^{r-1}$	Annulator polynomial of the support of $\boldsymbol{x}$.
$\boldsymbol{y}$	$\mathbb{F}_{q^m}^{n-k}$	Public syndrome.
$\boldsymbol{a}$	$\mathbb{F}_{q^m}^{r-1}$	Vector used in the MPC rank checking protocol.
$\boldsymbol{w}$	$\mathbb{F}_{q^m}^{r-1}$	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 $\mathbb{F}_{q^m} = \mathbb{F}_{2^m}$. We define $\mathbb{F}_{q^m}$ as $\mathbb{F}_2[X]/\langle P \rangle$ where P is a sparse irreducible polynomial of degree m. Table 2 describes the chosen polynomials.

m	P
31	$X^{31} + X^3 + 1$
37	$X^{37} + X^6 + X^4 + X + 1$
43	$X^{43} + X^6 + X^4 + X^3 + 1$

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

Elements of $\mathbb{F}_{q^m}$ are represented in the polynomial basis. An array of $l = \lceil \frac{m}{8} \rceil$ bytes $[b_1, \ldots, b_l]$ is used to store an element u of $\mathbb{F}_{q^m}$. Each byte b_i can be converted into a bitstring $\beta_1, \ldots, \beta_8$ where $\beta_j = (b_i >> j)$ & `0x1`.

Let $\beta_1, \ldots, \beta_{8l}$ be the bitstring stored in the byte array $[b_1, \ldots, b_l]$, then:

$$u = \sum_{i=1}^{m} \beta_i X^{i-1}$$

Converting a byte stream into an element of $\mathbb{F}_{q^m}$

When sampling randomness, we want to convert a byte stream into elements of $\mathbb{F}_{q^m}$. Sampling an element u is done is the following way:

- Obtain $l = \lceil \frac{m}{8} \rceil$ bytes from the stream. Let $\beta_1, \ldots, \beta_{8l}$ be the obtained bitstring.
- Set $u = \sum_{i=1}^{m} \beta_i X^{i-1}$.

To sample an array of $\mathbb{F}_{q^m}$ elements of length n we simply repeat this process n times, requiring $n \times \lceil \frac{m}{8} \rceil$ bytes `b` of randomness. We refer to this process as `FqmArrayFromBytes(b)`.

To convert an array $\boldsymbol{y}$ of n elements in $\mathbb{F}_{q^m}$ into a byte array, we simply revert this process, writing the values of the bits $\beta_1, \ldots, \beta_{8l}$ for each element of $\mathbb{F}_{q^m}$ and repeating the process n times. We refer to this process as `FqmArrayToBytes(`$\boldsymbol{y}$`)`.

Compacting arrays of $\mathbb{F}_{q^m}$ elements

In order to obtain a more compact representation (for values that will be sent over the network for example), we want to use a more compact representation for arrays of $\mathbb{F}_{q^m}$ elements.

Let $\boldsymbol{y}$ be an array of n $\mathbb{F}_{q^m}$ elements. Instead of representing each $\mathbb{F}_{q^m}$ element in a fixed number of bytes, we concatenate the n bitstring obtained from the y_i into a bitstring of

length nm. This process thus requires $\lceil \frac{nm}{8} \rceil$ bytes instead of $n \times \lceil \frac{m}{8} \rceil$ for representing the whole array.

We refer to these subroutines as `FqmArrayToString(`$\boldsymbol{y}$`)` and `FqmArrayFromString(str)` respectively.

Computing the weight of an array of $\mathbb{F}_{q^m}$ elements

The following algorithm computes the rank weight of an $\mathbb{F}_{q^m}$-vector.

Algorithm 1 `GetRankWeight`

Input: A vector $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$ of length n
Output: Its rank weight $r \in \mathbb{N}$

1: Initialize a new zero matrix $\boldsymbol{M} \in \mathbb{F}_q^{m \times n}$.
2: **for** $j \in [n]$ **do**
3: Fill the j-th column of $\boldsymbol{M}$ with the coefficients of $\boldsymbol{x}[j]$ seen as a polynomial.

$$\boldsymbol{x}[j] = \sum_{i=1}^{m} \boldsymbol{M}[i][j] X^{i-1}$$

4: **return** $\mathsf{Rank}(\boldsymbol{M})$

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 2 `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

The routines presented in this subsection sample from a seed some set of elements needed in the scheme.

Algorithm 3 `SampleFqmVector`

Input: Length n, a seed `seed`
Output: $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$

`PRG.Init(seed)`
`xbytes` = `PRG.GetBytes(`$n \times \lceil \frac{m}{8} \rceil$`)`
$\boldsymbol{x}$ = `FqmArrayFromBytes(xbytes)`
return $\boldsymbol{x}$

Algorithm 4 `SampleFqmMatrix`

Input: Dimensions $k \times n$, a seed `seed`
Output: $\boldsymbol{A} \in \mathbb{F}_{q^m}^{k \times n}$

`PRG.Init(seed)`
for $i \in [k]$ **do**
  `xbytes` = `PRG.GetBytes(`$n \times \lceil \frac{m}{8} \rceil$`)`
  $\boldsymbol{A}_{i,*}$ = `FqmArrayFromBytes(xbytes)`
return $\boldsymbol{A}$

The following algorithm samples a support of rank weight r, represented by a basis $\boldsymbol{supp} \in \mathbb{F}_{q^m}^r$. The support always contain the element $1 \in \mathbb{F}_{q^m}$, as explained in Section 3. The algorithm further samples a vector $\boldsymbol{x}$ of support $\boldsymbol{supp}$.

Algorithm 5 `SampleFqmVectorAndSupport`

Input: Length n, rank weight r, a seed `seed`
Output: $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$ of rank weight r and its support $\boldsymbol{supp} \in \mathbb{F}_{q^m}^r$

`PRG.Init(seed)`
`r_bytes` $= r\lceil \frac{m}{8} \rceil$
repeat
  $\boldsymbol{supp}[1] = 1$ ▷ The support always contains one
  $\boldsymbol{supp}[2, r]$ = `FqmArrayFromBytes(PRG.GetBytes(r_bytes))` ▷ Sampling the rest of the support
until `GetRankWeight`$(\boldsymbol{supp}) = r$
$\boldsymbol{x} = (0, \ldots, 0)$ ▷ Initialize $\boldsymbol{x}$ with zeros
repeat
  `rand` =`PRG.GetBytes`$(\lceil \frac{nr}{8} \rceil)$
  Convert `rand` into a bitstring `randbits` of length $8\lceil \frac{nr}{8} \rceil$
  for $i \in [n]$ **do**
    for $j \in [r]$ **do**
      $\boldsymbol{x}_i = \boldsymbol{x}_i +$ `randbits`$[ir + j] \cdot \boldsymbol{supp}[j]$
until `GetRankWeight`$(\boldsymbol{x}) = r$
return $\boldsymbol{x}, \boldsymbol{supp}$

Algorithm 6 `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**
  $\texttt{seed}^{(e)}$ = `PRG.GetBytes`(ℓ)
return $(\texttt{seed}^{(e)})_{e \in [\tau]}$

Algorithm 7 `SampleShares`

Input: A seed `seed`
Output: A set of shares $(\boldsymbol{x}_B, \boldsymbol{\beta}, \boldsymbol{a}, c) \in \mathbb{F}_{q^m}^k \times \mathbb{F}_{q^m}^{r-1} \times \mathbb{F}_{q^m}^{r-1} \times \mathbb{F}_{q^m}$

`PRG.Init(seed)`
`xbytes` = `PRG.GetBytes`$(k \times \lceil \frac{m}{8} \rceil)$
$\boldsymbol{x}_B$ = `FqmArrayFromBytes(xbytes)`
`bbytes` = `PRG.GetBytes`$((r-1) \times \lceil \frac{m}{8} \rceil)$
$\boldsymbol{\beta}$ = `FqmArrayFromBytes(bbytes)`
`abytes` = `PRG.GetBytes`$((r-1) \times \lceil \frac{m}{8} \rceil)$
$\boldsymbol{a}$ = `FqmArrayFromBytes(abytes)`
`cbytes` = `PRG.GetBytes`$(1 \times \lceil \frac{m}{8} \rceil)$
c = `FqmArrayFromBytes(cbytes)`$[0]$
return $\boldsymbol{x}$

Algorithm 8 `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(`$(n+1) \times \lceil \frac{m}{8} \rceil$`)`
  $\boldsymbol{\gamma}^{(e)}$ = `FqmArrayFromBytes(xbytes[..n])`
  $\epsilon^{(e)}$ = `FqmArrayFromBytes(xbytes[n+1])`$[0]$
return $(\boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})_{e\in[\tau]}$

Algorithm 9 `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)` $\mod N$
return $(i^{*(e)})_{e\in[\tau]}$

4.5 MPC routines

Algorithm 10 `MPC-RSD.ComputeAlpha`

Input:

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

Output: $\boldsymbol{\alpha}$

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

Algorithm 11 `MPC-RSD.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 matrix $\boldsymbol{H}$ and the public syndrome $\boldsymbol{y}$

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

1: **for** $\delta \in [D]$ **do**
2: $\quad \hat{\boldsymbol{x}}_A[\delta] = \boldsymbol{y} - \boldsymbol{H}\hat{\boldsymbol{x}}_B[\delta]$
3: $\quad \hat{\boldsymbol{x}}[\delta] = (\hat{\boldsymbol{x}}_A[\delta] \,||\, \hat{\boldsymbol{x}}_B[\delta])$
4: $\quad \hat{\boldsymbol{\alpha}}[\delta] = \texttt{MPC-RSD.ComputeAlpha}(\hat{\boldsymbol{x}}[\delta], \hat{\boldsymbol{a}}[\delta], (\gamma_j)_{j\in[1,n]}, \epsilon)$
5: $\quad \hat{z}[\delta] = -\sum_{j=1}^{n} \gamma_j(\hat{\boldsymbol{x}}_j[\delta]^{q^r} - \hat{\boldsymbol{x}}_j[\delta])$
6: $\quad \hat{v}[\delta] = \epsilon \cdot \hat{z}[\delta] - \langle \boldsymbol{\alpha}, \hat{\boldsymbol{\beta}}[\delta]\rangle - \hat{c}[\delta]$
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 12 `Hypercube.MainShares-Compute`

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

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

Algorithm 13 `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: $\quad \hat{\boldsymbol{\alpha}}[\delta] = \sum_{\substack{i\in[N] \\ \mathtt{bit}(i-1,\delta)=0}} \boldsymbol{\alpha}[i]$
3: $\quad \hat{v}[\delta] = \sum_{\substack{i\in[N] \\ \mathtt{bit}(i-1,\delta)=0}} v[i]$

4.7 Key parsing routines

Algorithm 14 `ParseSecretKey`

Input: Secret key `sk`
Output: $\boldsymbol{H}, \boldsymbol{x}, \boldsymbol{y}$ and the annihilator polynomial $\boldsymbol{\beta}$

1: $(\mathtt{sk\_seed}, \mathtt{pk\_seed}) = \mathtt{sk}$
2: $(\boldsymbol{x}, \boldsymbol{supp}) = \mathtt{SampleFqmVectorAndSupport(n,\ r,\ sk\_seed)}$
3: $\boldsymbol{H} = \mathtt{SampleFqmMatrix}(n-k, k, \mathtt{pk\_seed})$
4: $\boldsymbol{y} = (I|\boldsymbol{H})\boldsymbol{x}$
5: $\boldsymbol{\beta} = \mathtt{ComputeAnnihilatorPolynomial}(\boldsymbol{supp})$
6: **return** $(\boldsymbol{H}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\beta})$

Algorithm 15 `ParsePublicKey`

Input: Public key `pk`
Output: $\boldsymbol{H}, \boldsymbol{y}$

1: $(\mathtt{pk\_seed}, \mathtt{y\_string}) = \mathtt{pk}$
2: $\boldsymbol{H} = \mathtt{SampleFqmMatrix}(n-k, k, \mathtt{pk\_seed})$
3: $\boldsymbol{y} = \mathtt{FqmArrayFromString(y\_string)}$
4: **return** $(\boldsymbol{H}, \boldsymbol{y})$

4.8 PRG tree routines

Algorithm 16 `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 $(\mathtt{seed}_i)_{i\in[N]}$

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

Algorithm 17 `PRGPartialTreeReveal`

Input: A root seed `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 $(\texttt{seed}_i)_{i\in[D]}$

$(\texttt{seed\_left}, \texttt{seed\_right}) = \texttt{Hash}(\texttt{DS\_T}, \mathsf{salt}, e, \texttt{seed})$
if $D = 1$ **then**
  if $i^* = 1$ **then**
    return `seed_right`
  else
    return `seed_left`
else
  if $i^* \leq N/2$ **then**
    $(\texttt{seed}_i)_{i\in[1,D-1]} = \texttt{PRGPartialTreeReveal}(\texttt{seed\_left}, N/2, \mathsf{salt}, e, i^*)$
    $\texttt{seed}_D = \texttt{seed\_right}$
  else
    $\texttt{seed}_1 = \texttt{seed\_left}$
    $(\texttt{seed}_i)_{i\in[2,D]} = \texttt{PRGPartialTreeReveal}(\texttt{seed\_right}, N/2, \mathsf{salt}, e, i^* - N/2)$
  return $(\texttt{seed}_i)_{i\in[D]}$

Algorithm 18 `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 $(\texttt{seed}_i)_{i\in[N],i\neq i^*}$

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

4.9 Protocol specification

Algorithm 19 `RYDE_Keygen`

Input: Security level λ
Output: Secret key sk, public key pk

$\texttt{sk\_seed} = \texttt{RandomBytes}(\lambda)$
$\texttt{pk\_seed} = \texttt{RandomBytes}(\lambda)$
$(\boldsymbol{x}, \_) = \texttt{SampleFqmVectorAndSupport(n, r, sk\_seed)}$
$\boldsymbol{H} = \texttt{SampleFqmMatrix}(n-k, k, \texttt{pk\_seed})$
$\boldsymbol{y} = (I|\boldsymbol{H})\boldsymbol{x}$
$\mathsf{pk} = \texttt{pk\_seed} \,||\, \texttt{FqmArrayToString}(\boldsymbol{y})$
$\mathsf{sk} = \texttt{sk\_seed} \,||\, \texttt{pk\_seed}$
return sk, pk

Algorithm 20 RYDE_Sign

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

▷ **Step 0: Parse keys**
$(\boldsymbol{H}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\beta}) = \mathtt{ParseSecretKey}(\mathsf{sk})$
Define $\boldsymbol{x}_B$ as the last k coordinates of $\boldsymbol{x}$.

▷ **Step 1: Commitment**
$\mathsf{salt} = \mathtt{RandomBytes}(2\lambda)$
$\mathtt{mseed} = \mathtt{RandomBytes}(\lambda)$
$\mathtt{md} = \mathtt{Hash}(\mathtt{DS\_M}, m)$
$(\mathsf{seed}^{(e)})_{e\in[\tau]} = \mathtt{SampleSeeds}(\lambda, \tau, \mathtt{mseed})$
for $e \in [\tau]$ **do**
  $(\mathsf{seed}_i^{(e)})_{i\in[N]} = \mathtt{PRGTreeExpand}(\mathsf{seed}^{(e)}, N, \mathsf{salt}, e)$
  for $i \in [N-1]$ **do** ▷ Compute leaf shares
    $(\boldsymbol{x}_B^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i]) = \mathtt{SampleShares}(\mathsf{seed}_i^{(e)})$
    $\mathsf{cmt}_i^{(e)} = \mathtt{Hash}(\mathtt{DS\_C}, \mathsf{salt}, e, i, \mathsf{seed}_i^{(e)})$
  $\boldsymbol{a}^{(e)}[N] = \mathtt{SampleFqmVector}(\mathsf{seed}_N^{(e)}, r-1)$ ▷ Compute final leaf shares
  $\boldsymbol{a}^{(e)} = \sum_{i\in[N]} \boldsymbol{a}^{(e)}[i]$
  $\boldsymbol{x}_B^{(e)}[N] = \boldsymbol{x}_B - \sum_{i=1}^{N-1} \boldsymbol{x}_B^{(e)}[i]$
  $\boldsymbol{\beta}^{(e)}[N] = \boldsymbol{\beta} - \sum_{i=1}^{N-1} \boldsymbol{\beta}^{(e)}[i]$
  $c^{(e)}[N] = -\langle \boldsymbol{a}^{(e)}, \boldsymbol{\beta}\rangle - \sum_{i=1}^{N-1} c^{(e)}[i]$
  $\mathsf{cmt}_N^{(e)} = \mathtt{Hash}(\mathtt{DS\_C}, \mathsf{salt}, e, N, \mathsf{seed}_i^{(e)}, \boldsymbol{x}_B^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$
  $(\hat{\boldsymbol{x}}_B^{(e)}[\delta], \hat{\boldsymbol{\beta}}^{(e)}[\delta], \hat{\boldsymbol{a}}^{(e)}[\delta], \hat{c}^{(e)}[\delta])_{\delta\in[D]} = \mathtt{Hypercube.MainShares\text{-}Compute}((\boldsymbol{x}_B^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i])_{i\in[N]})$
$h_1 = \mathtt{Hash}(\mathtt{md}, \mathsf{pk}, \mathtt{salt}, \mathtt{DS\_1}, \mathsf{cmt}_1^{(1)}, \ldots, \mathsf{cmt}_N^{(1)}, \mathsf{cmt}_1^{(2)}, \ldots, \mathsf{cmt}_1^{(\tau)}, \cdots, \mathsf{cmt}_N^{(\tau)})$ ▷ Commit hypercube

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

▷ **Step 3: First response**
for $e \in [\tau]$ **do**
  $\boldsymbol{\alpha}^{(e)} = \mathtt{MPC\text{-}RSD.ComputeAlpha}(\boldsymbol{x}, \boldsymbol{a}^{(e)}, \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
  $(\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]} = \mathtt{MPC\text{-}RSD.Exec}\big[(\hat{\boldsymbol{x}}_B^{(e)}[\delta], \hat{\boldsymbol{\beta}}^{(e)}[\delta], \hat{\boldsymbol{a}}^{(e)}[\delta], \hat{c}^{(e)}[\delta])_{\delta\in[D]}, \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)}, \boldsymbol{\alpha}^{(e)}, \boldsymbol{H}, \boldsymbol{y}\big]$
$h_2 = \mathtt{Hash}(\mathtt{md}, \mathsf{pk}, \mathtt{salt}, h_1, \mathtt{DS\_2}, (\boldsymbol{\alpha}^{(e)}, (\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]})_{e\in[\tau]})$

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

▷ **Step 5: Second response**
for $e \in [\tau]$ **do**
  $\mathtt{ptree}^{(e)} = \mathtt{PRGPartialTreeReveal}(\mathsf{seed}^{(e)}, i^{*(e)})$
  $\boldsymbol{x}_A^{(e)}[i^{*(e)}] = \begin{cases} \boldsymbol{y} - \boldsymbol{H}\boldsymbol{x}_B^{(e)}[i^{*(e)}] & \text{if } i^{*(e)} = 1 \\ -\boldsymbol{H}\boldsymbol{x}_B^{(e)}[i^{*(e)}] & \text{otherwise} \end{cases}$
  $\boldsymbol{x}^{(e)}[i^{*(e)}] = (\boldsymbol{x}_A^{(e)}[i^{*(e)}] \,||\, \boldsymbol{x}_B^{(e)}[i^{*(e)}])$
  $\mathtt{MPC\text{-}RSD.ComputeAlpha}(\boldsymbol{x}^{(e)}[i^{*(e)}], \boldsymbol{a}^{(e)}[i^{*(e)}], \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
  if $i^{*(e)} = N$ **then**
    $\mathsf{rsp}^{(e)} = (\mathtt{ptree}^{(e)}, \mathsf{cmt}_{i^{*(e)}}^{(e)}, \boldsymbol{\alpha}^{(e)}[i^{*(e)}], \boldsymbol{0}, \boldsymbol{0}, 0)$
  else
    $\mathsf{rsp}^{(e)} = (\mathtt{ptree}^{(e)}, \mathsf{cmt}_{i^{*(e)}}^{(e)}, \boldsymbol{\alpha}^{(e)}[i^{*(e)}], \boldsymbol{x}_B^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$

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

Algorithm 21 `RYDE_Verify`

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

▷ **Step 0: Parse keys**
$(\boldsymbol{H}, \boldsymbol{y}) = \mathtt{ParsePublicKey}(\mathsf{pk})$

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

▷ **Step 2: Recompute h_1**
$\mathtt{md} = \mathtt{Hash}(\mathtt{DS\_M}, m)$
for $e \in [\tau]$ **do**
  $(\mathsf{seed}_i^{(e)})_{i\in[N], i\neq i^{*(e)}} = \mathtt{PRGPartialTreeExpand}(\mathsf{ptree}^{(e)}, N, \mathsf{salt}, e)$
  for $i \in [N], i \neq i^{*(e)}$ **do**
    if $i \neq N$ **then**
      $(\boldsymbol{x}_B^{(e)}[i], \boldsymbol{\beta}^{(e)}[i], \boldsymbol{a}^{(e)}[i], c^{(e)}[i]) = \mathtt{SampleShares}(\mathsf{seed}_i^{(e)})$
      $\mathsf{cmt}_i^{(e)} = \mathtt{Hash}(\mathtt{DS\_C}, \mathsf{salt}, e, i, \mathsf{seed}_i^{(e)})$
    else
      $\boldsymbol{a}^{(e)}[N] = \mathtt{SampleFqmVector}(\mathsf{seed}_N^{(e)}, r-1)$
      $\mathsf{cmt}_N^{(e)} = \mathtt{Hash}(\mathtt{DS\_C}, \mathsf{salt}, e, N, \mathsf{seed}_i^{(e)}, \boldsymbol{x}_B^{(e)}[N], \boldsymbol{\beta}^{(e)}[N], c^{(e)}[N])$
$\bar{h}_1 = \mathtt{Hash}(\mathtt{DS\_1}, \mathtt{md}, \mathsf{pk}, \mathtt{salt}, \mathsf{cmt}_1^{(1)}, \ldots, \mathsf{cmt}_N^{(1)}, \mathsf{cmt}_1^{(2)}, \ldots, \mathsf{cmt}_1^{(\tau)}, \cdots, \mathsf{cmt}_N^{(\tau)})$

▷ **Step 3: Recompute h_2**
for $e \in [\tau]$ **do**
  for $i \in [N], i \neq i^{*(e)}$ **do**
    $\boldsymbol{x}_A^{(e)}[i] = \begin{cases} \boldsymbol{y} - \boldsymbol{H}\boldsymbol{x}_B^{(e)}[i] & \text{if } i = 1 \\ -\boldsymbol{H}\boldsymbol{x}_B^{(e)}[i] & \text{otherwise} \end{cases}$
    $\boldsymbol{x}^{(e)}[i] = (\boldsymbol{x}_A^{(e)}[i] \,||\, \boldsymbol{x}_B^{(e)}[i])$
    $z^{(e)}[i] = -\sum_{j=1}^{n} \gamma_j^{(e)} (\boldsymbol{x}_j^{(e)}[i]^{q^r} - \boldsymbol{x}_j^{(e)}[i])$
    $\boldsymbol{\alpha}^{(e)}[i] = \mathtt{MPC\text{-}RSD.ComputeAlpha}(\boldsymbol{x}^{(e)}[i], \boldsymbol{a}^{(e)}[i], \boldsymbol{\gamma}^{(e)}, \epsilon^{(e)})$
  $\boldsymbol{\alpha}^{(e)} = \sum_i \boldsymbol{\alpha}^{(e)}[i]$
  for $i \in [N], i \neq i^{*(e)}$ **do**
    $v^{(e)}[i] = \epsilon^{(e)} \cdot z^{(e)}[i] - \langle \boldsymbol{\alpha}^{(e)}, \boldsymbol{\beta}^{(e)}[i] \rangle - c^{(e)}[i]$
  $v^{(e)}[i^{*(e)}] = -\sum_{i\neq i^{*(e)}} v^{(e)}[i]$
  $(\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]} = \mathtt{Hypercube.MainAlphaAndV\text{-}Compute}((\boldsymbol{\alpha}^{(e)}[i], v^{(e)}[i])_{i\in[N]})$
$\bar{h}_2 = \mathtt{Hash}(\mathtt{DS\_2}, \mathtt{md}, \mathsf{pk}, \mathtt{salt}, h_1, (\boldsymbol{\alpha}^{(e)}, (\hat{\boldsymbol{\alpha}}^{(e)}[\delta], \hat{v}^{(e)}[\delta])_{\delta\in[D]})_{e\in[\tau]})$

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

5 Parameter Sets

The Rank Syndrome Decoding problem has the following parameters:

- $q \in \mathbb{N}$ - the order of the base field;
- $m \in \mathbb{N}$ - the extension degree of the extension field $\mathbb{F}_{q^m}$;
- $n \in \mathbb{N}$ - the length of the code $\mathcal{C}$;
- $k \in \mathbb{N}$ - the dimension of the code;
- $r \in \mathbb{N}$ - the rank of the vector $\boldsymbol{x}$.

The other parameters of RYDE include:

- $N \in \mathbb{N}$ - the number of parties of the MPC protocol;
- $\tau \in \mathbb{N}$ - the number of parallel repetitions;
- $\eta \in \mathbb{N}$ - the extension degree for the MPC protocol (checking relations on $\mathbb{F}_{q^{m \cdot \eta}}$).

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

- the security of the Rank-SD 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 Additive-sharing scheme

The choice of the number $N \in \mathbb{N}$ of shares in the MPC protocol is a parameter which impacts performances: the bigger N, the slower the signing and verification, but the shorter the signature. We propose here two versions of the signature scheme with different values for N: we set $N = 256$ for a short version of the signature, and $N = 32$ for a fast version. There is also the number τ of repetitions required to reach the desired level of security, which increases the size and the signing time.

With additive sharing, we take $q = 2$ since it is the most efficient (and it does not impose a constraint on the number of parties unlike with threshold sharing). The choice of (q, m, n, k, r) was then made in order to have the necessary security in regards to the considered attacks. As for the choice of τ, we need to choose it such that our signature scheme resists to the forgery attack described in [KZ20].

We add a parameter η in the signature: the degree of the extension field on which the relations are checked in the MPC protocol. While it increases the quantity of information to transmit per round, it also increases the soundness of the zero-knowledge proof thus reducing the number of necessary repetitions τ. We choose for each set of parameters the optimal value of η with respect to the signature size.

Parameters are chosen based on the effectiveness of existing attacks on the Rank-SD problem (see section 9.2). 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`. The following table presents our different sets of secured parameters in the case of additive shares:

Instance	NIST Security Level	q	m	n	k	r	N	η	τ	sk	pk	σ
`RYDE-128F`	1	2	31	33	15	10	32	1	30	32 B	86 B	7.446 kB
`RYDE-128S`	1	2	31	33	15	10	256	1	20	32 B	86 B	5.956 kB
`RYDE-192F`	3	2	37	41	18	13	32	1	44	48 B	131 B	16.380 kB
`RYDE-192S`	3	2	37	41	18	13	256	1	29	48 B	131 B	12.933 kB
`RYDE-256F`	5	2	43	47	18	17	32	1	58	64 B	188 B	29.134 kB
`RYDE-256S`	5	2	43	47	18	17	256	1	38	64 B	188 B	22.802 kB

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

We refer to the design document for the computation of the theoretical sizes [BCDF+23].

The main difference from the theoretical sizes relies on the packing of the vector: RYDE implementation individually packs each vector in the signature instead of a single vector containing all the vector coefficients.

5.2 Threshold-sharing scheme

In [BCDF+23], two different variant of RYDE 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 Rank-SD problem (we obtain roughly the same signature sizes while not being particularly faster with threshold techniques). 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 RYDE.

Benchmark platform. The benchmarks have been performed on a machine running Ubuntu Server 22.04.2 LTS equipped with an Intel 13th Gen 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 11.3.0) 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 25 random instances. To minimize biases from background tasks running on the benchmark platform, each instance has been repeated 25 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
RYDE-128F	86.0 K	88.9 M	82.8 M
RYDE-128S	85.9 K	340.4 M	331.3 M
RYDE-192F	141.1 K	221.1 M	204.0 M
RYDE-192S	143.9 K	876.5 M	826.7 M
RYDE-256F	186.8 K	415.4 M	385.7 M
RYDE-256S	186.5 K	1500.3 M	1444.9 M

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

6.2 Optimized Implementation

The performance concerning the optimized 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
RYDE-128F	33.2 K	5.4 M	4.4 M
RYDE-128S	33.1 K	23.4 M	20.1 M
RYDE-192F	48.4 K	12.2 M	10.7 M
RYDE-129S	48.5 K	49.6 M	44.8 M
RYDE-256F	71.9 K	26.0 M	22.7 M
RYDE-256S	72.0 K	105.5 M	94.9 M

Table 5: Thousand (K) and Million (M) of CPU cycles of RYDE 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 Rank-SD instance. We expect our parameters to offer the security required for each security category. For the hardness of solving a Rank-SD instance, we refer to the following section, as well as the design document [BCDF$^+$23], where we describe the two best attacks on the Rank-SD problem, namely, the kernel attack and the Support Minor 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.

As for the proof of the unforgeability of the scheme, we have the following result in the EUF-CMA model.

Theorem 1. *Let the PRG used in the signature be (t, ϵ_{PRG})-secure and assume an adversary running in time t has advantage at most ϵ_{RSD} against the underlying Rank-SD problem. Let H_0, H_1, H_2, H_3, and H_4 be random oracles with output length 2λ bits. Let an adversary who makes q_0, q_1, q_2, q_3, q_4, q_S queries to the random oracles and signing oracle respectively. The probability that the adversary running in time at most t in producing an*

existential forgery under chosen message attack (EUF-CMA) is upper-bounded as:

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

where: τ *is the number of rounds of the signature,* $X = \max_{i \in [0,q_2]}\{X_i\}$ *where* $X_i \sim \mathcal{B}(\tau, p)$*,* $Y = \max_{i \in [0,q_4]}\{Y_i\}$ *where* $Y_i \sim \mathcal{B}(\tau - X, \frac{1}{N})$ *and* $q = \max\{q_0, q_1, q_2, q_3, q_4\}$*.*

We refer to the design rationale document for the proof of the theorem and the formal definition of the EUF-CMA security of a signature.

To summarize, we have an EUF-CMA scheme, relying on the hardness of solving the Rank-SD scheme. The parameters are taken such that:

- Solving the Rank-SD instance chosen respects the NIST security level aimed at;
- Building the signature without knowing the secret key respects the NIST security level aimed at;
- Pairs of messages and signatures 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 proofs for the security of the schemes, they are modeled as random oracles. In practice, we use `SHAKE` and `SHA3`, which are collision resistant and preimage resistant too. With Grover's algorithm, the complexity to find a preimage is of $\mathcal{O}(\frac{n}{2})$ (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. Hence, as in many other cryptosystems, we consider here that a hash function with an output of $2 \cdot \lambda$ bits provides a 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 considered parameters close to the Gilbert-Varshamov bound, combinatorial and algebraic attack behave similarly. So that in practice, resistance to quantum attacks may imply roughly dividing by a factor 2 the classical security.

9 Analysis of known attacks

9.1 Kales and Zaverucha's attack against Fiat-Shamir Signatures

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 brute-force one for protocols with more than one challenge, i.e. for protocols of a minimum of 5 rounds.

Kales and Zaverucha describe in [KZ20] an approach which consists in 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 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. Let C_1 and C_2 the respective cardinals to the first and second challenge sets, the probability P_1 of guessing at least r of the τ first-round challenges correctly is given by the equation:

$$P_1 = \sum_{k=r}^{\tau} \binom{\tau}{k} \left(\frac{1}{C_1}\right)^k \left(\frac{C_1 - 1}{C_1}\right)^{\tau - k}$$

The optimal number of repetitions to achieve the attack is:

$$\tau' = \arg \min_{0 \leq \tau \leq r} \left\{ \frac{1}{P_1} + C_2^{r-\tau'} \right\}$$

then the number of repetitions τ must be chosen such that this number allows to achieve the desired security.

In the case of the additive case, one obtains:

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

9.2 Combinatorial attacks against Rank-SD problem

We describe in this section the two most efficient combinatorial attacks against Rank-SD problem.

9.2.1 Enumeration of basis Chabaud and Stern proposed an algorithm in [CS96] which solve the problem by enumerating the possible supports for the vector $\boldsymbol{x}$. For each of them, one must translate the equations in $\mathbb{F}_{q^m}$ to equations in $\mathbb{F}_q$ as follows.

If trying all the different possible supports in Hamming metric is inefficient due to their large number (there are $\binom{n}{\omega}$ vectors of weight ω in $\mathbb{F}_2^n$), this is a viable method in rank metric: there are approximately $q^{(m-r)r}$ linear subspaces in $\mathbb{F}_{q^m}$ of dimension r. Moreover, one can always make the assumption that 1 is in the support, since it is enough to multiply by an invertible constant in $\mathbb{F}_{q^m}$ to make it appear in the coefficients of $\boldsymbol{x}$. There are approximately $q^{(m-r)(r-1)}$ linear subspaces in $\mathbb{F}_{q^m}$ which contains 1.

One obtains a system that has $(n-k)m$ independent equations and $nr + m$ variables in $\mathbb{F}_q$. Thanks to the gaussian elimination, this system can be solved with $O((nr+m)^3)$ operations in $\mathbb{F}_q$. It is deduced that the complexity of this attack is upper bounded by $O((nr+m)^3 q^{(m-r)(r-1)})$.

9.2.2 GRS Algorithm This method is an adaptation of the information set decoding attack used in Hamming metric. From the syndrome $\boldsymbol{y}$ of length $n-k$, the algorithm consists in guessing a set of $n-k$ coordinates which contains the support of the vector $\boldsymbol{x}$. One obtains $n-k$ equations and $n-k$ unknown coordinates of $\boldsymbol{x}$, which can be recovered by inverting an extracted matrix from H of size $(n-k)\times(n-k)$.

In order to adapt this method to coding theory in rank metric, Gaborit, Ruatta and Schrek proposed in [GRS13] to consider a linear subspace in $\mathbb{F}_{q^m}$ of dimension $r' \geq r$, and hope that includes the support of the vector $\boldsymbol{x}$.The probability that E of dimension r is included in E' of dimension r' (for $r' \geq r$) is $q^{-(m-r')r}$.

Suppose one knows a subspace E' of dimension r' which contains $E = \mathsf{Supp}(\boldsymbol{x})$. It is possible to recover $\boldsymbol{x}$ by solving a this linear system as long as: $r'n \leq (n-k)m$ i.e. $r' \leq \left\lfloor \dfrac{(n-k)m}{n} \right\rfloor$

Choosing the higher possible value $r = \lfloor \frac{(n-k)m}{n} \rfloor$, one gets the probability to obtain a suitable subspace E' equal to $q^{-(m-r')r} = q^{-r\lfloor \frac{km}{n} \rfloor}$. As in the previous attack, one can recover x by executing gaussian elimination, which can be perform in $O((n-k)^3m^3)$ operations in $\mathbb{F}_q$. This attack can be achieved with an average complexity: $O((n-k)^3m^3q^{r\lfloor \frac{km}{n} \rfloor})$.

9.2.3 An improvement of GRS Algorithm The idea proposed in [AGHT18] is, instead of consider a linear subspace in $\mathbb{F}_{q^m}$ which contains 1, choose a completely random subspace E'. If E' contains a subspace of the form αE, with $\alpha \in \mathbb{F}_{q^m}^*$, then one can retrieve the code word as before.

At the cost of an increase in the dimension of the code (k becomes $k+1$, since we multiply by an element α which does not belong to the code), we can obtain a new subspace that contains αE.

This attack can be achieved with an average complexity: $O((n-k)^\omega m^\omega q^{r\lfloor \frac{(k+1)m}{n} \rfloor - m})$, where ω is in practice equal to 2.

9.3 Algebraic attacks against Rank-SD problem

Algebraic attacks amount to solve the system of equations $\boldsymbol{Hx} = \boldsymbol{y}$ using computer algebra techniques like Gröbner basis. Today, the best algebraic modelings for solving the Rank-SD problem are the MaxMinors modeling [BBB$^+$20,BBC$^+$20] and the Support Minors modeling [BBC$^+$20,BBB$^+$22]. They can both benefits from a classical improvement that is the hybrid approach, that consists in specializing some variables with all possible values and solving the resulting system with less variables, but the same number of equations. In certain cases, the gain in complexity for solving the specialized system compared to the original one superseeds the lost in complexity coming from the exhausive search.

Hybrid methods As shown in [BBB$^+$22, Section 5], solving a Rank-SD problem of parameters (q,m,n,k,r) amount to solve q^{ar} smaller Rank-SD problems of parameters $(q,m,n-a,k-a,r)$.

MaxMinors Modeling Let $\boldsymbol{H}, \boldsymbol{x}, \boldsymbol{y}$ be a Rank-SD problem with parameters (q, m, n, k, r), i.e. $\boldsymbol{H} = (\boldsymbol{I}\ \boldsymbol{H'}) \in \mathbb{F}_{q^m}^{(n-k)\times n}$ is uniformly sampled, $\boldsymbol{x} \in \mathbb{F}_{q^m}^n$ has rank exactly[9] r, and $\boldsymbol{y} = \boldsymbol{H}\boldsymbol{x}$.

The idea of the MaxMinors modeling from [BBB+20] is the following: let $\boldsymbol{E} = (E_1, \ldots, E_r)$ be a basis of $\mathsf{Supp}(\boldsymbol{x})$, and $\boldsymbol{X} = (x_{i,j}) \in \mathbb{F}_q^{r\times n}$ a matrix representing the coordinates of $\boldsymbol{x}$ in the basis $\boldsymbol{E}$, then $\boldsymbol{x}^\top = (E_1, \ldots, E_r)\boldsymbol{X}$. Considering the extended code C' described in the previous section, and a parity-check $\boldsymbol{H'} \in \mathbb{F}_{q^m}^{(n-k-1)\times n}$ of this code, we have the relation :

$$\boldsymbol{H'}\boldsymbol{x} = 0, \text{ i.e. } (E_1, \ldots, E_r)\boldsymbol{X}\boldsymbol{H'}^\top = 0.$$

This implies that the matrix $\boldsymbol{X}\boldsymbol{H'}^\top$ is not full rank, and that all its maximal minors are equal to zero. By using the Cauchy-Binet formula that expresses the determinant of the product of two matrices $A \in \mathbb{K}^{r\times n}$ and $B \in \mathbb{K}^{n\times r}$ as

$$|AB| = \textstyle\sum_{T\subset\{1..n\},\#T=r} |A|_{*,T}\,|B|_{T,*}, \tag{2}$$

each of these maximal minors can be expressed as a linear combination of the maximal minors of the matrix $\boldsymbol{X}$. Using these minors as new variables, we obtain a system of $\binom{n-k-1}{r}$ linear equations in $\binom{n}{r}$ variables with coefficients over $\mathbb{F}_{q^m}$, that can be unfolded over $\mathbb{F}_q$ as $m\binom{n-k-1}{r}$ equations in the same number of variables (that are searched over $\mathbb{F}_q$).

It is possible to solve the Rank-SD problem with the MaxMinors modeling as soon as

$$\begin{cases} m\binom{n-k-1}{r} \geq \binom{n}{r} - 1, & \text{if the system has 1 solution,} \\ m\binom{n-k-1}{r} \geq \binom{n}{r}, & \text{if the system has no solution.} \end{cases} \tag{3}$$

It is possible to improve slightly the solving by puncturing the code on p positions, as long as

$$m\binom{n-p-k-1}{r} \geq \binom{n-p}{r} - 1. \tag{4}$$

With the largest such p, the linear system has almost the same number of equations than variables. If (3) is not satisfied, then we use the hybrid approach and specialize $a \in \{0..k\}$ columns of $\boldsymbol{X}$. The final complexity of the MaxMinors attack is bounded by

$$O\left(q^{ar}\binom{n-a-p}{r}^\omega\right) \tag{5}$$

as $n \to \infty$, where ω is the linear algebra constant. In all our estimates, we use conservatively $\omega = 2$, see Table 6.

We observe in Figures 3, 4, 5 that this approach is efficient for small r's, and becomes equivalent or worse than the combinatorial approach for r around the Gibert-Varshamov bound, as in this case a is close to k (see Figure 2).

[9] We assume the rank r is known. Otherwise, we can try to solve for $r = 1$, $r = 2$, ... until we find the good r.

NIST security level	q	m	n	k	r	combi	a	p	$\mathbb{C}_{MM}$	b	a	p	$\mathbb{C}_{SM+}$
1	2	31	33	15	10	**148**	12	2	153	1	11	0	155
3	2	37	41	18	13	**217**	15	2	238	2	14	0	241
5	2	43	47	18	17	**284**	15	2	308	-	-	-	317

Table 6: Complexity of the algebraic attacks for the proposed parameters. a is the hybrid parameter, p the number of columns on which the code is punctured. The complexities are given in $\log_2$ bit operations. We use $\omega = 2$.
- For the 256 bits of security parameters, the SM attack does not work: as it needs $a = k$, it amounts to a combinatorial attack

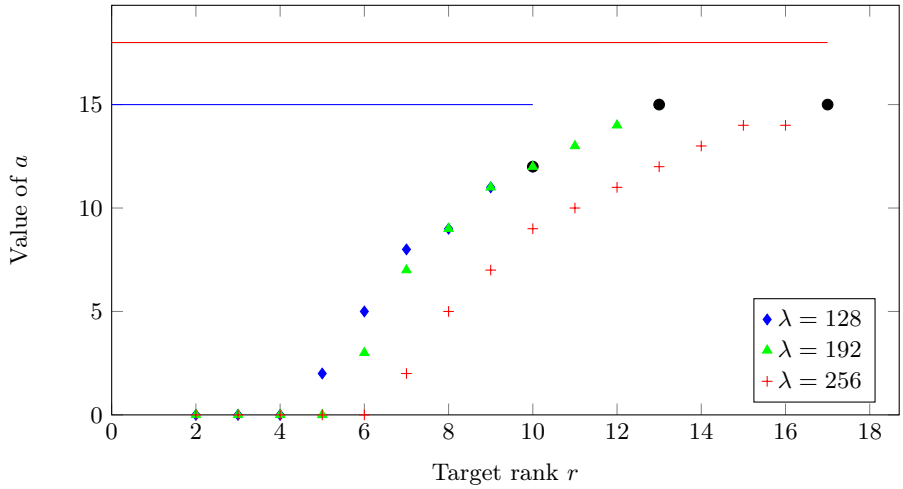

Fig. 2: Values of a for the MaxMinors modeling, in terms of $r \in \{0..r_{GV}\}$ below the Gibert-Varshamov bound. The values are given for the three proposed parameter sets (λ, m, n, k, r) equal to $(128, 31, 33, 15, 10), (192, 37, 41, 18, 13), (256, 43, 47, 18, 17)$ with $q = 2$. The horizontal lines represent the value of k. The black points correspond to the selected value of r for the proposed parameters.

The Support Minors modeling The previous modeling is a linearization technique, that only works with a large hybrid parameter a for large r. An alternative method is to rely on the Support Minors modeling, which was initially introduced in [BBC+20] to solve generic MinRank instances, and that uses a new set of variables. It has been adapted to instances coming from $\mathbb{F}_{q^m}$-linear codes in [BBB+22].

Write the received vector $\boldsymbol{v} = -\boldsymbol{mG} + \boldsymbol{x}$ with $-\boldsymbol{mG} \in C$ a codeword, and $\mathsf{Rank}(\boldsymbol{x}) = r$. Then, $\boldsymbol{x} = \boldsymbol{v} + \boldsymbol{mG} = (E_1, \ldots, E_r)\boldsymbol{C}$ so that any row $\boldsymbol{r}_i$ of $\boldsymbol{M}(\boldsymbol{v} + \boldsymbol{mG}) \in \mathbb{F}_q^{m \times n}$ is in the row space of $\boldsymbol{C}$. Therefore, all the maximal minors of the matrix $\binom{\boldsymbol{r}_i}{\boldsymbol{C}}$ are equal to 0. This system is described and analyzed in [BBB+22], where it is shown that it contains the previous equations from the MaxMinors modeling.

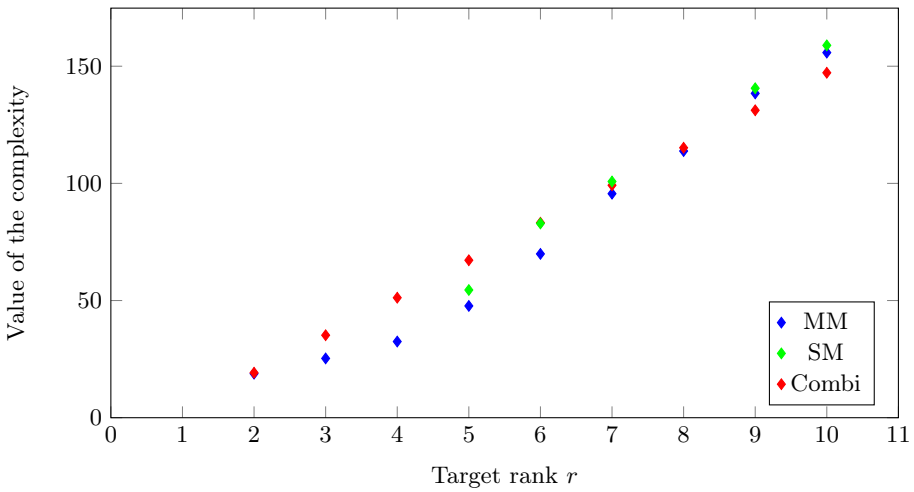

Fig. 3: Comparison of the MaxMinors complexity and the combinatorial attacks for the parameters $\lambda = 128$, $(2, 31, 33, 15)$ and $r \in \{2..10\}$. Only the values of the complexities with $a < k$ are ploted.

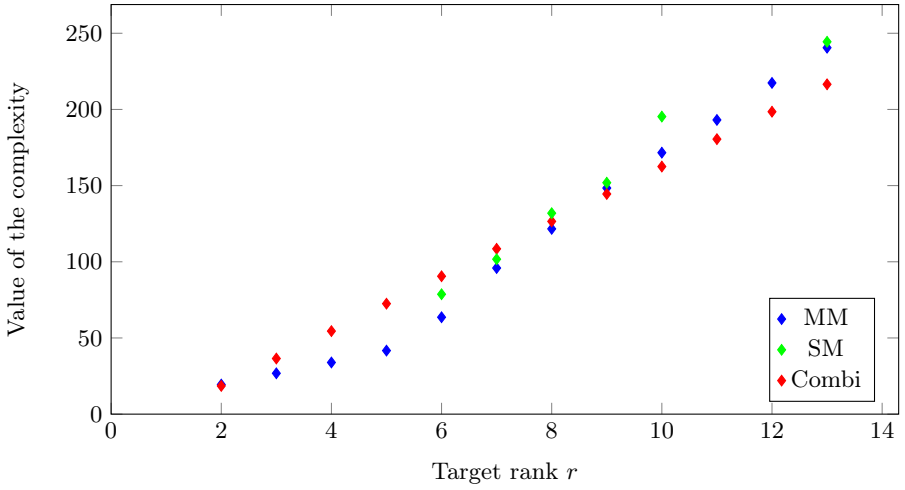

Fig. 4: Comparison of the MaxMinors complexity and the combinatorial attacks for the parameters $\lambda = 192$, $(2, 37, 41, 18)$ and $r \in \{1..13\}$. Only the values of the complexities with $a < k$ are ploted.

It is conjectured in [BBB+22], based on a careful theoretical analysis of the system and some experimental heuristics, that it is possible to solve this bilinear system by linear

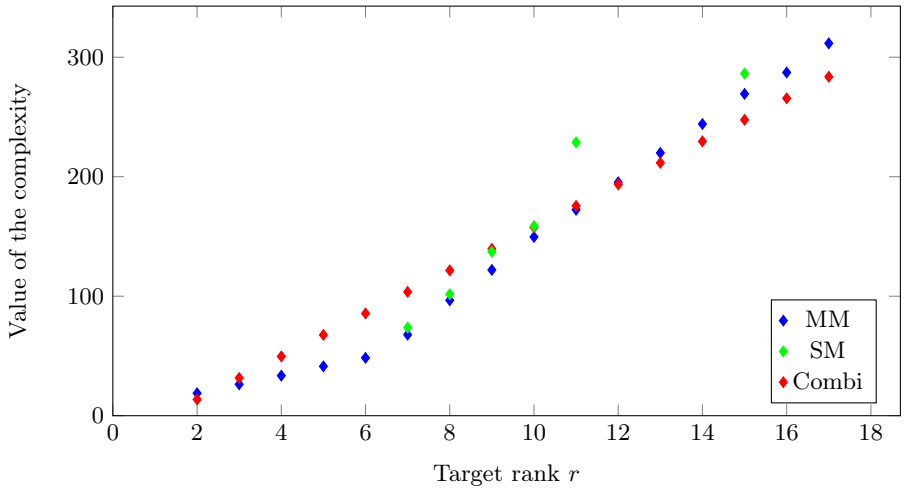

Fig. 5: Comparison of the MaxMinors complexity and the combinatorial attacks for the parameters $\lambda = 256$, $(2, 43, 47, 18)$ and $r \in \{1..17\}$. Only the values of the complexities with $a < k$ are ploted.

algebra on a matrix with N rows and M columns, as soon as $N \geq M - 1$, with:

$$N = \sum_{i=1}^{k} \binom{n-i}{r}\binom{k+b-1-i}{b-1} - \binom{n-k-1}{r}\binom{k+b-1}{b} \tag{6}$$

$$- (m-1)\sum_{i=1}^{b}(-1)^{i+1}\binom{k+b-i-1}{b-i}\binom{n-k-1}{r+i}. \tag{7}$$

$$M = \binom{k+b-1}{b}\left(\binom{n}{r} - m\binom{n-k-1}{r}\right). \tag{8}$$

In this case, the final cost in $\mathbb{F}_q$ operations is given by

$$\mathcal{O}\left(m^2 N M^{\omega-1}\right),$$

where ω is the linear algebra constant and where the m^2 factor comes from expressing each $\mathbb{F}_{q^m}$ operation involved in terms of $\mathbb{F}_q$ operations. As previously, it is possible to use an hybrid approach to reach the constraint $N \geq M - 1$, or to puncture the code to get N and M of the same value. The complexity estimates for the parameters sets are ploted in Figures 3, 4, 5. We can see that for $q = 2$, the MaxMinors modeling is (almost) always better than the Support Minors one.

10 Advantages and limitations

10.1 Advantages

Difficulty of the underlying problem: the security of the signature scheme relies on the genuine problem of decoding in rank metric Rank-SD, and there exists a probabilistic reduction from a generic NP-complete problem to the Rank-SD problem [GZ14]. The

Rank-SD problem has been studied for many years. In particular the security of the problem corresponds to parameters on the Rank-Gilbert-Varshamov bound, the hardest area for parameters in which recent algebraic attacks [BBC+20] behave similarly to classical combinatorial attacks.

Size of public keys and signature: the size of our parameters remains low. When comparing to Dilithium level 1, the sum of public key and signature sizes of RYDE is less than 60% higher than Dilithium (the sum for Dilithium reaches 3.7kB compared to 5.9kB for RYDE). It makes RYDE one of the shortest MPCitH signatures.

Size of the public matrix: in our case the public matrix can be generated from a random seed. Now independently of this, the properties of rank metric make the size of the public matrix very small (and this without additional cyclic structure), for instance of order 1kB for the first level of parameters: it could be an advantage if one were to choose not to generate the public matrix from a seed.

No cyclic structure of the underlying problem: our security is based on a problem which does not rely on cyclic structure for which the quantum security is not fully known.

Resilience against Rank-SD attacks: the size of the signature is composed of two parts: a part related to the MPC that only depends on the security level (seeds, hashes) and a part related to the parameters of the chosen RSD instance. Hence, increasing the size of the problem parameters has a limited impact on the total size of the signature. For example, the RYDE-128S instance features a signature size of 5.9kB using the parameters $(q, m, n, k, r) = (2, 31, 29, 14, 10)$ for NIST security level 1. Choosing the parameters $(q, m, n, k, r) = (2, 37, 38, 16, 14)$ would reach more than 192 bits of security for the underlying problem and result in a signature size of 6.9 kB for NIST security level 1. Thus, if one discovers an efficient attack against the Rank-SD problem that forces us to increase the problem parameters, only the problem parameter part will be impacted and the overall effect on the signature length will be mitigated, as explained in the previous example.

10.2 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 when the security level increases, both the parameters of the Rank-SD instance and the number of repetitions tau need to be increased. For example, going from security level 1 to 3, the number of repetitions is increased by half (20 to 30) while the quantity of data to be transmitted per turn grows as well since the parameters of the Rank-SD instance also have to be increased. Notice that this limitation is coherent with the Rank-SD resilience advantage from Section 10.1. Indeed in that case only the instance parameters need to be increased and not the number of repetitions nor the size of seeds and hashes.

Signature speed: Compared to lattice based signatures, the speed of our scheme is slower even if it compares very well with other signature schemes in general.

Complex implementation: this scheme is relatively complex to implement. Although it has potential to perform well on hardware, its implementation could be difficult in embedded systems.

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. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 418–433, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[AGHT18] Nicolas Aragon, Philippe Gaborit, Adrien Hauteville, and Jean-Pierre Tillich. A New Algorithm for Solving the Rank Syndrome Decoding Problem. In *ISIT 2018 - IEEE International Symposium on Information Theory*, pages 2421–2425, Vail, United States, June 2018.

[AMGH+22] Carlos Aguilar-Melchor, Nicolas Gama, James Howe, Andreas Hülsing, David Joseph, and Dongze Yue. The Return of the SDitH. *Cryptology ePrint Archive, Report 2022/1645*, 2022.

[BBB+20] Magali Bardet, Pierre Briaud, Maxime Bros, Philippe Gaborit, Vincent Neiger, Olivier Ruatta, and Jean-Pierre Tillich. An algebraic attack on rank metric code-based cryptosystems. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 64–93, Cham, 2020. 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, 2022.

[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, International Conference on the Theory and Application of Cryptology and Information Security, 2020. Proceedings*, pages 507–536, 2020.

[BCDF+23] Loïc Bidoux, Jesús-Javier Chi-Domínguez, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Matthieu Rivain, and Adrien Vinçotte. RYDE : A Digital Signature Scheme based on Rank-Syndrome-Decoding problem with MPCitH paradigm. arXiv, 2023.

[BGRV23] Loïc Bidoux, Philippe Gaborit, Olivier Ruatta, and Adrien Vinçotte. MPCitH-based Signature for the RSD problem using a Hypercube. In *IEEE International Symposium on Information Theory (ISIT)*, 2023.

[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.

[CS96] Florent Chabaud and Jacques Stern. The Cryptographic Security of the Syndrome Decoding Problem for Rank Distance Codes. In *Advances in Cryptology - ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security, Kyongju, Korea, November 3-7, 1996, Proceedings*, volume 1163 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 1996.

[Fen22] Thibauld Feneuil. Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP. *Cryptology ePrint Archive, Report 2022/1512*, 2022.

[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.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *International Cryptology Conference (CRYPTO)*, 1986.

[GRS13] Philippe Gaborit, Olivier Ruatta, and Julien Schrek. On the complexity of the Rank Syndrome Decoding problem, 2013.

[GZ14] Philippe Gaborit and Gilles Zémor. On the hardness of the decoding and the minimum distance problems for rank codes, 2014.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-Knowledge from Secure Multiparty Computation. In *Proceedings of the 39th annual ACM symposium on Theory of computing (STOC)*, 2007.

[KZ20] Daniel Kales and Greg Zaverucha. An Attack on Some Signature Schemes Constructed From Five-Pass Identification Schemes. In *International Conference on Cryptology and Network Security (CANS)*, 2020.

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