Name of the Proposal:

MiRitH (MinRank in the Head)

June 2023

Inventors of the Scheme:

Gora Adj[1], Luis Rivera-Zamarripa[1], Javier Verbel[1]

Additional Designers:

Emanuele Bellini[1], Stefano Barbero[2], Andre Esser[1],
Carlo Sanna[2†], Floyd Zweydinger[1]

[1]Cryptography Research Center, Technology Innovation Institute
Masdar City, P.O. Box 9639, Abu Dhabi, UAE

{gora.adj,emanuele.bellini,andre.esser,
luis.zamarripa,javier.verbel,floyd.zweydinger}@tii.ae

[2]Group of Cryptography and Number Theory
Department of Mathematical Scpackes, Politecnico di Torino
Corso Duca degli Abruzzi 24, 10129 Turin, Italy

{stefano.barbero,carlo.sanna}@polito.it

[†]C. Sanna was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

Contents

List of Tables

List of Figures

1 Introduction

MiRitH (from MinRank-in-the-Head) is a digital signature scheme based on the hardness of the MinRank problem. Informally, the MinRank problem asks to find a non-trivial low-rank linear combination of some given matrices over a finite field. The construction of MiRitH starts from an MPC-in-the-Head (MPCitH) based Zero-Knowledge Proof of Knowledge (ZKPoK) of a solution to the MinRank problem, which is used to construct a 5-pass identification scheme. This identification protocol is then converted into a non-interactive signature scheme via the Fiat–Shamir transform.

1.1 Basic approach

In this section, we provide a rather informal overview of the techniques used by MiRitH.

The signature scheme is obtained from a 5-pass identification protocol via the Fiat–Shamir transform. In turn, the identification protocol follows the MPCitH paradigm. Therefore, to identify itself to a verifier, the prover simulates (in his head) an MPC protocol for N parties to prove knowledge about the solution to the MinRank problem. Later the prover reveals some states of this MPC execution to the verifier that allow him to check the validity of the execution and the result. The general phases of the identification protocol are outlined in Fig. 1.

Inputs: Prover and verifier know the problem instance, prover holds a witness for a solution to the problem.

Phase 1: *Input preparation.* The prover prepares all inputs for a total of N parties of an MPC protocol and commits to their initial states.

Phase 2: *First Challenge.* The verifier sends a first challenge, which later affects the computations within the MPC protocol.

Phase 3: *MPC execution.* The prover simulates all MPC parties based on their initial inputs and the first challenge. Subsequently, he commits to the intermediate and final states of the executions.

Phase 4: *Second Challenge.* The verifier sends a challenge $i^* \in \{1, \ldots, N\}$.

Phase 5: *Reveal Phase.* The prover reveals the initial states of all MPC parties except for the i^*-th. Additionally, all intermediate states and the final state of party i^* are revealed.

Figure 1: Basic phases of the identification protocol of MiRitH.

Eventually, after receiving the initial states, the verifier can recompute all subsequent states of the MPC parties and check them against the commitments, for all parties except for the i^*-th party. Note that the verifier must not obtain knowledge of all initial states, since this would allow him to compute the solution to the MinRank problem. In general, this exclusion leads to a soundness probability of $1/N$ as guessing this challenge would allow cheating in the input preparation of the i^*-th party.[1] Also, by obtaining all the final states of the parties, the verifier can check that the MPC protocol execution was successful.

Concrete instantiations of the general MPCitH framework from Fig. 1, like MiRitH, are mainly characterized by the underlying problem and the used MPC protocol. Therefore, let us briefly

[1]The effective soundness probability is slightly larger, which is related to the probability of guessing the first challenge. However, the first challenge space is sufficiently large such that the probability is exponentially close to $1/N$.

sketch those characteristics for MiRitH.

1.1.1 Witness for a MinRank solution

We use the Kipnis–Shamir modeling [KS99], which essentially allows us to prove the knowledge of a MinRank solution $\boldsymbol{\alpha}$ by a matrix-product identity

$$\boldsymbol{M}_0 = \boldsymbol{M}_1 \cdot K, \tag{1}$$

where $\boldsymbol{M}_0$ and $\boldsymbol{M}_1$ are specifically crafted linear combinations of the matrices defining the problem instance depending on the solution $\boldsymbol{\alpha}$, and K is another secret matrix. Put differently, if $\boldsymbol{M}_0$ and $\boldsymbol{M}_1$ are correctly constructed, proving that Eq. (1) holds is equivalent to proving knowledge to a solution to the MinRank problem. More details on this modeling are given in Section 2.1.1. The main advantage of using this modeling is that we obtain a matrix identity, which can be verified via an MPC protocol instead of a statement about the rank of a matrix.

1.1.2 MPC protocol

MiRitH uses a linear secret sharing scheme between N parties, where each party i holds an additive share $\boldsymbol{\alpha}_i$ of the solution $\boldsymbol{\alpha} = \sum_{i=1}^{N} \boldsymbol{\alpha}_i$ and an additive share K_i of the matrix $K = \sum_{i=1}^{N} K_i$. Based on its shares, each party can construct its own share of $\boldsymbol{M}_0$ and $\boldsymbol{M}_1$. More details on additive sharings are provided in Section 2.

From here, an MPC protocol for the verification of matrix-multiplication triples is used. A matrix-multiplication triple is a set of matrices (A, B, C) fulfilling $A \cdot B = C$.

MiRitH uses this MPC protocol to prove that the shares of all parties form a valid matrix-multiplication triple, i.e., that they satisfy Eq. (1) and, hence, that $\boldsymbol{\alpha}$ forms a solution to the MinRank problem.

1.2 Comparison to other MinRank-based schemes

MiRitH is built on top of the MinRank-based signature scheme proposed by Adj, Rivera-Zamarripa, and Verbel [ARZV22]. In that work, Adj et al. introduce a Multi-Party Computation (MPC) protocol Π to verify solutions to the MinRank problem using the Kipnis–Shamir modeling [KS99] and checking that a triple of shared matrices (Z, X, Y) satisfies $Z = X \cdot Y$.

MiRitH introduces two optimizations over [ARZV22], namely:

1. It improves the protocol Π by employing an optimization analogous to the one introduced by Kales and Zarevucha [KZ22, Section 2.5]. Note that this optimization is also used in [Fen22];

2. It improves the protocol Π by reducing the size of a random matrix (later called R) used in the protocol, leveraging an optimization introduced by Feneuil [Fen22].

In the following we list other MinRank-based signature schemes in the literature and briefly explain how they differ from MiRitH.

- Courtois' scheme [Cou01]: To the best of our knowledge, this is the first signature scheme based on the hardness of the MinRank problem and already has a ring signature variant. Courtois' scheme, contrary to MiRitH, is derived directly from the Fiat–Shamir transform applied to a zero-knowledge sigma protocol with soundness error 2/3.

- MR-DSS [BESV22]: MR-DSS improves upon Courtois' scheme by using a sigma protocol with helper, which has a soundness error of 1/2. Also, the authors of MR-DSS formalized the ring signature version of the scheme. Our extension of MiRitH to ring signatures uses their formalization.
- Feneuil's schemes [Fen22]: Recently, Feneuil proposed two MinRank-based schemes, both using the MPCitH approach applied to an MPC protocol Π. These schemes differ from MiRitH in how they use Π to prove a solution to the MinRank problem.
  1. The first version uses the protocol Π to prove the knowledge of a linear combination of the given matrices that equals the product of two matrices, where the left factor has r columns, and the other one has r rows. In this case, the resulting product has the same dimension as the input matrices. MiRitH instead uses the Kipnis–Shamir modeling, which is slightly more efficient as it involves a matrix product with r fewer columns.
  2. The second protocol proposed by Feneuil instead proves that every row of the low-rank linear combination matrix represents an element of a field extension that is a root of a secret polynomial. This approach is completely different from that of MiRitH.

2 Notation

We list below some basic notations used throughout the paper:

- λ denotes the security parameter.
- q denotes the size of the field over which the MiRitH is defined.
- m denotes the number of rows of one matrix in the public key.
- n denotes the number of columns of one matrix in the public key.
- k denotes the size of the solution vector $\boldsymbol{\alpha}$ of the underlying MinRank problem.
- r denotes the target rank of the underlying MinRank problem.
- s denotes the number of rows of the matrices in the first challenges.
- D denotes the dimension of the hypercube.
- N denotes the number of parties in the MPC protocol.
- τ denotes the number of rounds.
- $[i] := \{1, 2, \ldots, i\}$ for any positive integer i.
- $\mathbb{F}_q$ denotes a finite field of q elements.
- $\mathbb{F}_q^{n_r \times n_c}$ is the vector space of $n_r \times n_c$ matrices over $\mathbb{F}_q$.
- $\mathsf{Rank}(M)$ is the rank of a matrix M.
- $\mathbf{0}$ is a zero matrix (the size will always be clear from the context).

- $\boldsymbol{M}$ denotes a tuple of $k+1$ matrices $(M_0, M_1, \ldots, M_k) \in (\mathbb{F}_q^{m\times n})^{k+1}$.
- $\boldsymbol{M_\alpha} := M_0 + \sum_{i=1}^k \alpha_i M_i$ for each $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k) \in \mathbb{F}_q^k$.
- $a \leftarrow \mathcal{A}(x)$ means that a is the output of an algorithm $\mathcal{A}$ on input x.
- $a \xleftarrow{\$} \mathcal{S}$ means that a is sampled uniformly at random from a set $\mathcal{S}$.
- log is the logarithm in base 2.
- NULL is a pointer value that does not point to any valid data object.
- $\emptyset$ denotes an empty string.
- string1||string2 denotes the concatenation of the strings string1 and string2.
- string[i] denotes the ith byte of the string of bytes string.

We also give the following relevant definitions:

- **Left- and Right-Side of a Matrix.** Let $A \in \mathbb{F}_q^{m\times n}$, and $1 \leq r < n$ be an integer. We define the *left-side* $A^{L_r} \in \mathbb{F}_q^{m\times(n-r)}$ and the *right-side* $A^{R_r} \in \mathbb{F}_q^{m\times r}$ of A, with respect to r, as the matrices satisfying
$$A = [A^{L_r} | A^{R_r}].$$
In the remainder of this specification, r always refers to the target rank of our MinRank instances. Hence, we write conveniently A^L and A^R.

- **Additive sharing.** Let a be an element in an additive group. We say that a tuple $[\![a]\!] := ([\![a]\!]_1, \ldots, [\![a]\!]_N)$ such that $a = \sum_{i=1}^N [\![a]\!]_i$ is an *additive sharing* of a.

- **Distributed operations in Multi-Party Computation.** Let $\mathbb{V}$ be a vector space over a field $\mathbb{F}$. In the MPC context, an additive sharing is distributed between N parties, i.e., each party gets one of the N shares. Thus the parties can do distributed computation on the shares $[\![x]\!]$, $[\![y]\!]$ of $x, y \in \mathbb{V}$ as follows:
  - **Addition.** The parties locally compute $[\![x+y]\!]$ by adding their respective shares:
$$[\![x+y]\!]_i := [\![x]\!]_i + [\![y]\!]_i \quad \forall i \in [N].$$
This process is denoted by $[\![x+y]\!] = [\![x]\!] + [\![y]\!]$.
  - **Addition with a constant.** For a given constant $a \in \mathbb{V}$ the parties locally compute $[\![x+a]\!]$ as
$$[\![x+a]\!]_1 = [\![x]\!]_1 + a \quad \text{and} \quad [\![x+a]\!]_i = [\![x]\!]_i \quad \text{for } i \in [N] \setminus \{1\}.$$
This process is denoted by $[\![x+a]\!] = [\![x]\!] + a$.
  - **Multiplication by a constant.** For a given constant $a \in \mathbb{F}$ the parties locally compute $[\![a \cdot x]\!]$ as
$$[\![a \cdot x]\!]_i = a \cdot [\![x]\!]_i \quad \forall i \in [N].$$
This process is denoted by $[\![a \cdot x]\!] = a \cdot [\![x]\!]$.

- **Matrix-multiplication triple.** Let X, Y, Z be matrices such that $Z = X \cdot Y$. Then, we say that the triple of shares $([\![X]\!], [\![Y]\!], [\![Z]\!])$ is a *matrix-multiplication triple.*

2.1 MinRank

The MinRank problem is the underlying hard problem of the signature scheme that we propose.

Problem (MinRank)**.** *Let q,m,n,k, and r be positive integers, with q a prime power. The MinRank problem with parameters (q, m, n, k, r) is defined as:*

Given: *$(k+1)$-tuple $\boldsymbol{M} = (M_0, M_1, \ldots, M_k) \in (\mathbb{F}_q^{m\times n})^{k+1}$.*

Find: *$\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k) \in \mathbb{F}_q^k$ such that $\mathsf{Rank}\big(M_0 + \sum_{i=1}^k \alpha_i M_i\big) \leq r$.*

2.1.1 Kipnis–Shamir modeling

The Kipnis–Shamir modeling was introduced in 1999 [KS99] and is based on the following fact. If there is a vector $\boldsymbol{\alpha} \in \mathbb{F}_q^k$ and a matrix $K \in \mathbb{F}_q^{r\times(n-r)}$ such that

$$\left(M_0 + \sum_{i=1}^{k} \alpha_i M_i\right) \cdot \begin{bmatrix} I \\ -K \end{bmatrix} = \mathbf{0}, \tag{2}$$

where $I \in \mathbb{F}_q^{(n-r)\times(n-r)}$ is a non-singular matrix, then the vector $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_k)$ is a solution to the instance $\boldsymbol{M} = (M_0, M_1, \ldots, M_k)$ of the MinRank problem.

If in Eq. (2) we fix I to be the identity matrix of size $(n-r) \times (n-r)$, then we get that

$$M_0^L + \sum_{i=1}^{k} \alpha_i M_i^L = \left(M_0^R + \sum_{i=1}^{k} \alpha_i M_i^R\right) \cdot K, \tag{3}$$

that is, $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$. To solve an instance of the MinRank problem, Kipnis and Shamir proposed to solve the bilinear system having $\alpha_1, \ldots, \alpha_k$ and the entries of K as unknowns and the entries of Eq. (2) as equations.

3 Signature Scheme

3.1 Design rationale

MiRitH is based on the MPCitH paradigm. Therefore, it uses an MPC protocol to prove the knowledge of a solution to the MinRank problem.

3.1.1 MPC protocol

As discussed in Section 2.1.1, in our proof of knowledge, the witness of a solution to an instance $\boldsymbol{M}$ of the MinRank problem is a pair $(\boldsymbol{\alpha}, K)$ consisting of a vector $\boldsymbol{\alpha} \in \mathbb{F}_q^k$ and a matrix $K \in \mathbb{F}_q^{r\times(n-r)}$ such that $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$. Let s be a positive integer. In Fig. 2, we describe an MPC protocol Π_s that checks the matrix identity $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$ with soundness error $1/q^s$ (see Proposition 2).

```
Inputs: Each party knows the MinRank instance $\boldsymbol{M}$ and takes a share of the following
sharings as input: $[\![\boldsymbol{\alpha}]\!]$ and $[\![K]\!]$, where $\boldsymbol{\alpha} \in \mathbb{F}_q^k$ and $K \in \mathbb{F}_q^{r\times(n-r)}$; $[\![A]\!]$, where $A$ has been
uniformly sampled from $\mathbb{F}_q^{s\times r}$; and $[\![C]\!]$, where $C = A \cdot K$.

MPC Protocol:
1 : The parties locally compute $[\![\boldsymbol{M}_{\boldsymbol{\alpha}}^L]\!]$ and $[\![\boldsymbol{M}_{\boldsymbol{\alpha}}^R]\!]$.
2 : The parties get a random matrix $R \xleftarrow{\$} \mathbb{F}_q^{s\times m}$.
3 : The parties locally set $[\![S]\!] = R \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^R]\!] + [\![A]\!]$.
4 : The parties open $[\![S]\!]$ so that they all obtain $S$.
5 : The party locally compute $[\![V]\!] = S \cdot [\![K]\!] - R \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^L]\!] - [\![C]\!]$.
6 : The parties open $[\![V]\!]$ to obtain $V$.
7 : The parties output accept if $V = \boldsymbol{0}$ and reject otherwise.
```

Figure 2: The MPC protocol Π_s to check that $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$.

3.1.2 ZKPoK for MinRank

Starting from the MPC protocol Π_s, we build a ZKPoK of a solution to an instance of the MinRank problem as described in Fig. 3. This ZKPoK employs a commitment scheme Com, a pseudorandom generator PRG, and a tree seed TreePRG (see Section 9.1 for details). The completeness, soundness, and honest-verifier zero-knowledge properties of the ZKPoK of Fig. 3 are provided in Section 9.

3.1.3 Non-interactive via the Fiat–Shamir transform

To transform the ZKPoK of Fig. 3 into a non-interactive signature scheme, we use a standard generalization of the Fiat–Shamir transform for canonical 5-pass protocols [FS87].

Precisely, for a message $\mathsf{msg} \in \{0,1\}^*$, the prover sample a random value $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$ and then simulates τ rounds of the protocol of Fig. 3 as follows. For each round $\ell \in [\tau]$, the prover computes the commitments $\mathsf{com}_1^{(\ell)}, \ldots, \mathsf{com}_N^{(\ell)}$ as in the interactive protocol, and then produces the hash

$$h_1 = \mathrm{Hash}_1\Big(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}_i^{(\ell)})_{i\in[N],\ell\in[\tau]}\Big)$$

and the first challenges $R^{(1)}, \ldots, R^{(\tau)} \leftarrow \mathrm{PRG}(h_1)$. Then, for each round $\ell \in [\tau]$, the prover computes the matrices $[\![S^{(\ell)}]\!]_1, [\![V^{(\ell)}]\!]_1, \ldots, [\![S^{(\ell)}]\!]_N, [\![V^{(\ell)}]\!]_N$ as in the interactive protocol. After that, the prover produces the hash

$$h_2 = \mathrm{Hash}_2\left(\mathsf{msg}, \mathsf{salt}, h_1, \left([\![S^{(\ell)}]\!]_i, [\![V^{(\ell)}]\!]_i\right)_{i\in[N],\ell\in[\tau]}\right)$$

and the second challenges $i^{*,(1)}, \ldots, i^{*,(\tau)} \leftarrow \mathrm{PRG}(h_2)$. Finally, the prover uses the challenges $(i^{*,(\ell)})_{\ell\in[\tau]}$ to assemble the signature

$$\sigma = \left(\mathsf{salt}, h_1, h_2, \left((\mathsf{state}_i^{(\ell)})_{i\neq i^{*,(\ell)}}, \mathsf{com}_{i^{*,(\ell)}}^{(\ell)}, [\![S^{(\ell)}]\!]_{i^{*,(\ell)}}\right)_{\ell\in[\tau]}\right)$$

```
$\mathsf{ZKProof}(\mathsf{Prover}(\boldsymbol{M}, \boldsymbol{\alpha}, K),\ \mathsf{Verifier}(\boldsymbol{M}))$
**Phase 1:** $\mathsf{Prover}$ sets up the inputs for the MPC protocol $\Pi_s$:
  1 : $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda$,   $(\mathsf{seed}_i, \rho_i)_{i \in [N]} \leftarrow \mathrm{TreePRG}(\mathsf{seed})$
  2 : For each party $i \in [N-1]$
        $[\![\boldsymbol{\alpha}]\!]_i, [\![K]\!]_i, [\![A]\!]_i, [\![C]\!]_i \leftarrow \mathrm{PRG}(\mathsf{seed}_i)$
        $\mathsf{state}_i \leftarrow \mathsf{seed}_i$
  3 : $[\![\boldsymbol{\alpha}]\!]_N \leftarrow \boldsymbol{\alpha} - \sum_{i \neq N} [\![\boldsymbol{\alpha}]\!]_i$,   $[\![K]\!]_N \leftarrow K - \sum_{i \neq N} [\![K]\!]_i$
  4 : $[\![A]\!]_N \leftarrow \mathrm{PRG}(\mathsf{seed}_N)$,   $[\![C]\!]_N \leftarrow A \cdot K - \sum_{i \neq N} [\![C]\!]_i$
  5 : $\mathsf{aux} \leftarrow ([\![\boldsymbol{\alpha}]\!]_N, [\![K]\!]_N, [\![C]\!]_N)$,   $\mathsf{state}_N \leftarrow (\mathsf{seed}_N, \mathsf{aux})$
  6 : Commit to each party's state: $\mathsf{com}_i \leftarrow \mathsf{Com}(\mathsf{state}_i, \rho_i)$, for all $i \in [N]$.
  7 : $\mathsf{Prover}$ computes $h_1 \leftarrow \mathrm{Hash}_1(\mathsf{com}_1, \ldots, \mathsf{com}_N)$ and sends it to $\mathsf{Verifier}$.
**Phase 2:** $\mathsf{Verifier}$ samples $R \xleftarrow{\$} \mathbb{F}_q^{s \times m}$ and sends it to $\mathsf{Prover}$.
**Phase 3:** $\mathsf{Prover}$ simulates the MPC protocol $\Pi_s$:
  8 : The parties locally compute $[\![\boldsymbol{M}^L_{\boldsymbol{\alpha}}]\!]$ and $[\![\boldsymbol{M}^R_{\boldsymbol{\alpha}}]\!]$.
  9 : The parties locally set $[\![S]\!] = R \cdot [\![\boldsymbol{M}^R_{\boldsymbol{\alpha}}]\!] + [\![A]\!]$.
  10 : The parties open $[\![S]\!]$ so that they all obtain $S$.
  11 : The parties locally compute $[\![V]\!] = S \cdot [\![K]\!] - R \cdot [\![\boldsymbol{M}^L_{\boldsymbol{\alpha}}]\!] - [\![C]\!]$.
  12 : $\mathsf{Prover}$ computes $h_2 \leftarrow \mathrm{Hash}_2([\![S]\!]_1, [\![V]\!]_1, \ldots, [\![S]\!]_N, [\![V]\!]_N)$.
  13 : $\mathsf{Prover}$ sends $h_2$ to $\mathsf{Verifier}$.
**Phase 4:** $\mathsf{Verifier}$ samples $i^* \xleftarrow{\$} [N]$ and sends it to $\mathsf{Prover}$.
**Phase 5:** $\mathsf{Prover}$ sends $\mathsf{rsp} := ((\mathsf{state}_i, \rho_i)_{i \neq i^*}, \mathsf{com}_{i^*}, [\![S]\!]_{i^*})$ to $\mathsf{Verifier}$.
**Verification:**
  14 : $\mathsf{Verifier}$ recompute $([\![S]\!]_i, [\![V]\!]_i)_{i \neq i^*}$ from $(\mathsf{state}_i)_{i \neq i^*}$.
  15 : $[\![V]\!]_{i^*} \leftarrow -\sum_{i \neq i^*} [\![V]\!]_i$
  16 : $\mathsf{Verifier}$ accepts if and only if $\mathsf{com}_i = \mathsf{Com}(\mathsf{state}_i, \rho_i)$, for each $i \neq i^*$,
       $h_1 = \mathrm{Hash}_1(\mathsf{com}_1, \ldots, \mathsf{com}_N), h_2 = \mathrm{Hash}_2([\![S]\!]_1, [\![V]\!]_1, \ldots, [\![S]\!]_N, [\![V]\!]_N)$.
```

Figure 3: Zero-knowledge proof of knowledge for MinRank.

which consists in revealing all the states of each round ℓ except for the $i^{*,(\ell)}$-th state, for which $\mathsf{com}^{(\ell)}_{i^{*,(\ell)}}$ and $[\![S^{(\ell)}]\!]_{i^{*,(\ell)}}$ are provided instead.

3.2 Key generation algorithm

The uncompressed public key and secret key of our scheme are a random instance $\boldsymbol{M}$ of the MinRank problem and the corresponding witness $(\boldsymbol{\alpha}, K)$, respectively. If stored directly, they would require $mn(k+1)\log q$ bits and $(k + r(n-r))\log q$ bits, respectively. We employ the key generation

```
KeyGen()
1 : $(\mathsf{seed}_{\mathsf{pk}}, \mathsf{seed}_{\mathsf{sk}}) \xleftarrow{\$} \{0,1\}^\lambda \times \{0,1\}^\lambda$
2 : $M_1, \ldots, M_k \leftarrow \mathsf{PRG\_matricesPK}(\mathsf{seed}_{\mathsf{pk}})$    // $M_1, \ldots, M_k \in \mathbb{F}_q^{m \times n}$
3 : $\boldsymbol{\alpha}, K, E^R \leftarrow \mathsf{PRG\_matricesSK}(\mathsf{seed}_{\mathsf{sk}})$    // $\boldsymbol{\alpha} \in \mathbb{F}_q^k$, $K \in \mathbb{F}_q^{r \times (n-r)}$, $E^R \in \mathbb{F}_q^{m \times r}$
4 : $E \leftarrow E^R \cdot K \mid E^R$
5 : $M_0 \leftarrow E - \sum_{i=1}^{k} \alpha_i M_i$
6 : $\mathsf{pk} \leftarrow (\mathsf{seed}_{\mathsf{pk}}, M_0)$
7 : $\mathsf{sk} \leftarrow \mathsf{seed}_{\mathsf{sk}}$
    return $(\mathsf{pk}, \mathsf{sk})$
```

```
DecompressPK(pk)
1 : $\mathsf{seed}_{\mathsf{pk}}, M_0 \leftarrow \mathsf{pk}$
2 : $M_1, \ldots, M_k \leftarrow \mathsf{PRG\_matricesPK}(\mathsf{seed}_{\mathsf{pk}})$
    return $(M_0, \ldots, M_k)$
```

```
DecompressSK(sk)
1 : $\mathsf{seed}_{\mathsf{sk}} \leftarrow \mathsf{sk}$
2 : $\boldsymbol{\alpha}, K, E^R \leftarrow \mathsf{PRG\_matricesSK}(\mathsf{seed}_{\mathsf{sk}})$
    return $(\boldsymbol{\alpha}, K)$
```

Figure 4: Algorithms for the generation and the decompression of the public and secret keys.

scheme proposed by Adj et al. [ARZV22], which compresses the public key and the secret key into $\lambda + mn \log q$ and λ bits, respectively. See Fig. 4 for the details.

We remark that using the key generation scheme proposed by Di Scala and Sanna [DSS23] the public key could be compressed further to $\lambda + (m(n-r) - k) \log q$ bits, but at the cost of more computation time for the key generation.

3.3 Signing and verification algorithms

The signing algorithm is derived as the non-interactive version of the identification protocol outlined in Section 3.1.2 by applying the Fiat–Shamir transform (compare to Section 3.1.3). Thereby, to reach the necessary soundness, we use τ parallel repetitions. Pseudocodes are given by Fig. 5 and Fig. 6.

3.4 Hypercube variant

The hypercube-MPCitH protocol, introduced by Aguilar-Melchor et al. [AMGH+23], allows one to reduce the number of parties to be simulated in an N^D-party classical MPCitH to only ND, without loss of soundness. To apply the hypercube approach to MiRitH, we set the number of

```
Sign($\boldsymbol{M}, \boldsymbol{\alpha}, K, \mathsf{msg}$)
1 :  $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
Phase 1: Set up the views for the MPC protocols
     for $\ell \in [\tau]$ do
2 :      $\mathsf{seed}^{(\ell)} \xleftarrow{\$} \{0,1\}^{\lambda}$,   $(\mathsf{seed}_i^{(\ell)})_{i \in [N]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{seed}^{(\ell)})$
         for $i \in [N-1]$ do
3 :          $[\![A^{(\ell)}]\!]_i, [\![\boldsymbol{\alpha}^{(\ell)}]\!]_i, [\![C^{(\ell)}]\!]_i, [\![K^{(\ell)}]\!]_i \leftarrow \mathsf{PRG\_shares}(\mathsf{salt}, \mathsf{seed}_i^{(\ell)}, i, N)$
4 :          $\mathsf{state}_i^{(\ell)} \leftarrow \mathsf{seed}_i^{(\ell)}$
5 :      $[\![A^{(\ell)}]\!]_N \leftarrow \mathsf{PRG\_shares}(\mathsf{salt}, \mathsf{seed}_N^{(\ell)}, N, N)$,   $[\![\boldsymbol{\alpha}^{(\ell)}]\!]_N \leftarrow \boldsymbol{\alpha} - \sum_{i \neq N} [\![\boldsymbol{\alpha}^{(\ell)}]\!]_i$
6 :      $[\![K^{(\ell)}]\!]_N \leftarrow K - \sum_{i \neq N} [\![K^{(\ell)}]\!]_i$,   $[\![C^{(\ell)}]\!]_N \leftarrow A^{(\ell)} \cdot K - \sum_{i \neq N} [\![C^{(\ell)}]\!]_i$
7 :      $\mathsf{aux}^{(\ell)} \leftarrow ([\![\boldsymbol{\alpha}^{(\ell)}]\!]_N, [\![K^{(\ell)}]\!]_N, [\![C^{(\ell)}]\!]_N)$,   $\mathsf{state}_N^{(\ell)} \leftarrow (\mathsf{seed}_N^{(\ell)}, \mathsf{aux}^{(\ell)})$
8 :      $\mathsf{com}_i^{(\ell)} \leftarrow \mathrm{Hash}_0\left(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)}\right)$, for all $i \in [N]$
Phase 2: First challenges
9 :  $h_1 \leftarrow \mathrm{Hash}_1\left(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}_i^{(\ell)})_{i \in [N], \ell \in [\tau]}\right)$
10 : $R^{(1)}, \ldots, R^{(\tau)} \leftarrow \mathsf{PRG\_first\_challenge}(h_1)$
Phase 3: Simulation of the MPC protocols
     for $\ell \in [\tau]$ do
11 :     Compute $[\![\boldsymbol{M}_{\boldsymbol{\alpha}}^{L,(\ell)}]\!], [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^{R,(\ell)}]\!]$ from $[\![\boldsymbol{\alpha}^{(\ell)}]\!]$
12 :     $[\![S^{(\ell)}]\!] \leftarrow R^{(\ell)} \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^{R,(\ell)}]\!] + [\![A^{(\ell)}]\!]$
13 :     $S^{(\ell)} \leftarrow \sum_i [\![S^{(\ell)}]\!]_i$
14 :     $[\![V^{(\ell)}]\!] \leftarrow S^{(\ell)} \cdot [\![K^{(\ell)}]\!] - R^{(\ell)} \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^{L,(\ell)}]\!] - [\![C^{(\ell)}]\!]$
Phase 4: Second challenges
15 : $h_2 \leftarrow \mathrm{Hash}_2\left(\mathsf{msg}, \mathsf{salt}, h_1, \left([\![S^{(\ell)}]\!]_i, [\![V^{(\ell)}]\!]_i\right)_{i \in [N],\, \ell \in [\tau]}\right)$
16 : $i^{*,(1)}, \ldots, i^{*,(\tau)} \leftarrow \mathsf{PRG\_second\_challenge}(h_2, N)$
Phase 5: Assembling the signature $\sigma$
17 : $\sigma \leftarrow \left(\mathsf{salt}, h_1, h_2, \left((\mathsf{state}_i^{(\ell)})_{i \neq i^{*,(\ell)}}, \mathsf{com}_{i^{*,(\ell)}}^{(\ell)}, [\![S^{(\ell)}]\!]_{i^{*,(\ell)}}\right)_{\ell \in [\tau]}\right)$
return $\sigma$
```

Figure 5: Signing algorithm of MiRitH.

parties to N^D, instead of N as in classical MiRitH. Then, the views for the N^D parties, called *leaf parties*, are set up and committed as in Phase 1 of Fig. 5. Next, each party $i \in [N^D]$ is expressed

```
Verif$(\boldsymbol{M}, \mathsf{msg}, \sigma)$
 1 : $R^{[1]}, \dots, R^{(\tau)} \leftarrow \mathsf{PRG\_first\_challenge}(h_1)$
 2 : $i^{*,[1]}, \dots, i^{*,(\tau)} \leftarrow \mathsf{PRG\_second\_challenge}(h_2, N)$
     for all $\ell \in [\tau]$ do
 3 :     Compute $\left[\!\left[V^{(\ell)}\right]\!\right]_i$ as in Sign
 4 :     Compute $\mathsf{com}_i^{(\ell)}$ and $\left[\!\left[S^{(\ell)}\right]\!\right]_i$ as in Sign, for all $i \in [N] \backslash \{i^{*,(\ell)}\}$
 5 :     $S^{(\ell)} \leftarrow \sum_i \left[\!\left[S^{(\ell)}\right]\!\right]_i$
 6 :     Compute $\left[\!\left[V^{(\ell)}\right]\!\right]_i$ as in Sign, for all $i \in [N] \backslash \{i^{*,(\ell)}\}$
 7 :     $\left[\!\left[V^{(\ell)}\right]\!\right]_{i^{*,(\ell)}} \leftarrow -\sum_{i \neq i^{*,(\ell)}} \left[\!\left[V^{(\ell)}\right]\!\right]_i$
 8 : $h_1' \leftarrow \mathrm{Hash}_1\big(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}_i^{(\ell)})_{i \in [N], \ell \in [\tau]}\big)$
 9 : $h_2' \leftarrow \mathrm{Hash}_2\big(\mathsf{msg}, \mathsf{salt}, h_1, (\left[\!\left[S^{(\ell)}\right]\!\right]_i, \left[\!\left[V^{(\ell)}\right]\!\right]_i)_{i \in [N],\, \ell \in [\tau]}\big)$
10 : Output accept if $h_1' = h_1$ and $h_2' = h_2$, otherwise output reject
```

Figure 6: Verification algorithm of MiRitH.

as $(i_1, \dots, i_D) \in [N]^D$ [2], and for every $k \in [D]$, we form N extra parties $\{(k, j) : j \in [N]\}$, called *main parties*. For a shared value X, the share $[\![X]\!]_{(k,j)}$ of a main party (k, j) will be the sum of the shares $[\![X]\!]_i$ of the leaf parties $i = (i_1, \dots, i_D)$ having $i_k = j$. Now, the MPC simulation in Phase 3 of Fig. 5 is carried out here on each subset $\{(k, j) : j \in [N]\}$ $(k \in D)$ of the ND main parties. The hypercube variant of MiRitH, which we call MiRitH-Hypercube, is described in Fig. 7 and Fig. 8.

The hypercube approach offers a two-fold advantage. First, the computational complexity is improved, since, besides having to simulate exponentially fewer parties as mentioned above, the N^D shares to set up can be essentially performed offline. Second, while executing D MPC protocols of size N each, only one auxiliary state is used for all of them, hence considerably reducing the communication cost. For further details and security analysis on the hypercube-MPCitH, we refer to the paper of Aguilar-Melchor et al. [AMGH+23].

[2] We represent i as $(i_1, \dots, i_D)$, where the i_j's are obtained by expressing $i - 1$ in N-ary as $i - 1 = i_1' + i_2' N + i_3' N^2 + \dots i_N' N^{D-1}$ and setting $i_j = i_j' + 1$ for every $j \in [N]$

```
$\mathsf{SignHypercube}(\boldsymbol{M}, \boldsymbol{\alpha}, K, \mathsf{msg})$
1 :  $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}$
Phase 1: Set up the views for the MPC protocols
     for $\ell \in [\tau]$ do
2 :    $\mathsf{seed}^{(\ell)} \xleftarrow{\$} \{0,1\}^{\lambda}$,   $(\mathsf{seed}_i^{(\ell)})_{i \in [N^D]} \leftarrow \mathrm{TreePRG}(\mathsf{salt}, \mathsf{seed}^{(\ell)})$
       for $(k, j) \in [D] \times [N]$ do
3 :      Initialize all entiries of $\llbracket A^{(\ell)} \rrbracket_{(k,j)}, \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,j)}, \llbracket C^{(\ell)} \rrbracket_{(k,j)}, \llbracket K^{(\ell)} \rrbracket_{(k,j)}$ to zero.
       for $i \in [N^D - 1]$ do
4 :      $\llbracket A^{(\ell)} \rrbracket_i, \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_i, \llbracket C^{(\ell)} \rrbracket_i, \llbracket K^{(\ell)} \rrbracket_i \leftarrow \mathsf{PRG\_shares}(\mathsf{salt}, \mathsf{seed}_i^{(\ell)}, i, N^D)$,   $\mathsf{state}_i^{(\ell)} \leftarrow \mathsf{seed}_i^{(\ell)}$
5 :      $(i_1, \ldots, i_D) \leftarrow i$   Note: Representation of $i$ in $[N]^D$
         for $k \in [D]$ do
6 :        $\llbracket A^{(\ell)} \rrbracket_{(k,i_k)} \leftarrow \llbracket A^{(\ell)} \rrbracket_{(k,i_k)} + \llbracket A^{(\ell)} \rrbracket_i$,   $\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,i_k)} \leftarrow \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,i_k)} + \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_i$,
7 :        $\llbracket C^{(\ell)} \rrbracket_{(k,i_k)} \leftarrow \llbracket C^{(\ell)} \rrbracket_{(k,i_k)} + \llbracket C^{(\ell)} \rrbracket_i$,   $\llbracket K^{(\ell)} \rrbracket_{(k,i_k)} \leftarrow \llbracket K^{(\ell)} \rrbracket_{(k,i_k)} + \llbracket K^{(\ell)} \rrbracket_i$
8 :    $\llbracket A^{(\ell)} \rrbracket_{N^D}, \leftarrow \mathsf{PRG\_shares}(\mathsf{salt}, \mathsf{seed}_{N^D}^{(\ell)}, N^D, N^D)$,   $\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{N^D} \leftarrow \boldsymbol{\alpha} - \sum_{i \neq N^D} \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_i$
9 :    $\llbracket K^{(\ell)} \rrbracket_{N^D} \leftarrow K - \sum_{i \neq N^D} \llbracket K^{(\ell)} \rrbracket_i$,   $\llbracket C^{(\ell)} \rrbracket_{N^D} \leftarrow A^{(\ell)} \cdot K - \sum_{i \neq N^D} \llbracket C^{(\ell)} \rrbracket_i$
       for $k \in [D]$ do
10 :     $\llbracket A^{(\ell)} \rrbracket_{(k,N)} \leftarrow \llbracket A^{(\ell)} \rrbracket_{(k,N)} + \llbracket A^{(\ell)} \rrbracket_{N^D}$,   $\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,N)} \leftarrow \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,N)} + \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{N^D}$,
11 :     $\llbracket C^{(\ell)} \rrbracket_{(k,N)} \leftarrow \llbracket C^{(\ell)} \rrbracket_{(k,N)} + \llbracket C^{(\ell)} \rrbracket_{N^D}$,   $\llbracket K^{(\ell)} \rrbracket_{(k,N)} \leftarrow \llbracket K^{(\ell)} \rrbracket_{(k,N)} + \llbracket K^{(\ell)} \rrbracket_{N^D}$
12 :   $\mathsf{aux}^{(\ell)} \leftarrow (\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{N^D}, \llbracket K^{(\ell)} \rrbracket_{N^D}, \llbracket C^{(\ell)} \rrbracket_{N^D})$,   $\mathsf{state}_{N^D}^{(\ell)} \leftarrow (\mathsf{seed}_{N^D}^{(\ell)}, \mathsf{aux}^{(\ell)})$
13 :   $\mathsf{com}_i^{(\ell)} \leftarrow \mathrm{Hash}_0\,(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)})$,  for all $i \in [N^D]$
14 :   $\mathsf{com}^{(\ell)} \leftarrow \mathrm{Hash}_{(1,0)}\big(\mathsf{salt}, \ell, (\mathsf{com}_i^{(\ell)})_{i \in [N^D]}\big)$
Phase 2: First challenges
15 :  $h_1 \leftarrow \mathrm{Hash}_{(1,1)}\big(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}^{(\ell)})_{\ell \in [\tau]}\big)$
16 :  $R^{(1)}, \ldots, R^{(\tau)} \leftarrow \mathsf{PRG\_first\_challenge}(h_1)$
Phase 3: Simulation of the MPC protocols
     for $\ell \in [\tau]$ do
       for $k \in [D]$ do
         for $j \in [N]$ do
17 :       Compute $\llbracket \boldsymbol{M}_{\boldsymbol{\alpha}}^{L,(\ell)} \rrbracket_{(k,j)}, \llbracket \boldsymbol{M}_{\boldsymbol{\alpha}}^{R,(\ell)} \rrbracket_{(k,j)}$ from $\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_{(k,j)}$
18 :       $\llbracket S^{(\ell)} \rrbracket_{(k,j)} \leftarrow R^{(\ell)} \cdot \llbracket \boldsymbol{M}_{\boldsymbol{\alpha}}^{R,(\ell)} \rrbracket_{(k,j)} + \llbracket A^{(\ell)} \rrbracket_{(k,j)}$
19 :     $S_k^{(\ell)} \leftarrow \sum_{j \in [N]} \llbracket S^{(\ell)} \rrbracket_{(k,j)}$
20 :     $\llbracket V^{(\ell)} \rrbracket_{(k,j)} \leftarrow S_k^{(\ell)} \cdot \llbracket K^{(\ell)} \rrbracket_{(k,j)} - R^{(\ell)} \cdot \llbracket \boldsymbol{M}_{\boldsymbol{\alpha}}^{L,(\ell)} \rrbracket_{(k,j)} - \llbracket C^{(\ell)} \rrbracket_{(k,j)}$,  for all $j \in [N]$
21 :     $H_k^{(\ell)} \leftarrow \mathrm{Hash}_{(2,0)}\big(\mathsf{salt}, \ell, (\llbracket S^{(\ell)} \rrbracket_{(k,j)}, \llbracket V^{(\ell)} \rrbracket_{(k,j)})_{j \in [N]}\big)$
Phase 4: Second challenges
22 :  $h_2 \leftarrow \mathrm{Hash}_{(2,1)}\big(\mathsf{msg}, \mathsf{salt}, h_1, (H_1^{(\ell)}, \ldots, H_D^{(\ell)})_{\ell \in [\tau]}\big)$
23 :  $i^{*,(1)}, \ldots, i^{*,(\tau)} \leftarrow \mathsf{PRG\_second\_challenge}(h_2, N^D)$,  where each $i^{*,(\ell)} \in [N^D]$
Phase 5: Assembling the signature $\sigma$
24 :  $\sigma \leftarrow \Big(\mathsf{salt}, h_1, h_2, \big((\mathsf{state}_i^{(\ell)})_{i \neq i^{*,(\ell)}}, \mathsf{com}_{i^{*,(\ell)}}^{(\ell)}, \llbracket S^{(\ell)} \rrbracket_{i^{*,(\ell)}}\big)_{\ell \in [\tau]}\Big)$
return $\sigma$
```

Figure 7: Signing algorithm of MiRitH-Hypercube.

```
VerifHypercube($\boldsymbol{M}$, msg, $\sigma$)
1 :  $R^{[1]}, \ldots, R^{(\tau)} \leftarrow \mathsf{PRG\_first\_challenge}(h_1), \quad i^{*,[1]}, \ldots, i^{*,(\tau)} \leftarrow \mathsf{PRG\_second\_challenge}(h_2, N^D)$
     for all $\ell \in [\tau]$ do
2 :      $\mathsf{com}_i^{(\ell)} \leftarrow \mathrm{Hash}_0\left(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)}\right)$, for all $i \in [N^D]\backslash\{i^{*,(\ell)}\}$
3 :      $\mathsf{com}^{(\ell)} \leftarrow \mathrm{Hash}_{(1,0)}\left(\mathsf{salt}, \ell, (\mathsf{com}_i^{(\ell)})_{i\in[N^D]}\right)$
4 :      $(i_1^{*,(\ell)}, \ldots, i_D^{*,(\ell)}) \leftarrow i^{*,(\ell)}$
         for $k \in [D]$ do
5 :          Compute $\left[\!\left[S^{(\ell)}\right]\!\right]_{(k,j)}$ as in SignHypercube, for all $j \in [N]\backslash\{i_k^{*,(\ell)}\}$
6 :          $S_k^{(\ell)} \leftarrow \sum_{j\in[N]} \left[\!\left[S^{(\ell)}\right]\!\right]_{(k,j)}$
7 :          Compute $\left[\!\left[V^{(\ell)}\right]\!\right]_{(k,j)}$ as in SignHypercube, for all $j \in [N]\backslash\{i_k^{*,(\ell)}\}$
8 :          $\left[\!\left[V^{(\ell)}\right]\!\right]_{(k,i_k^{*,(\ell)})} \leftarrow -\sum_{j\neq i_k^{*,(\ell)}} \left[\!\left[V^{(\ell)}\right]\!\right]_{(k,j)}$
9 :          $H_k^{(\ell)} \leftarrow \mathrm{Hash}_{(2,0)}\left(\mathsf{salt}, \ell, \left(\left[\!\left[S^{(\ell)}\right]\!\right]_{(k,j)}, \left[\!\left[V^{(\ell)}\right]\!\right]_{(k,j)}\right)_{j\in[N]}\right)$
10 : $h'_1 \leftarrow \mathrm{Hash}_{(1,1)}\left(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}^{(\ell)})_{\ell\in[\tau]}\right)$
11 : $h'_2 \leftarrow \mathrm{Hash}_{(2,1)}\left(\mathsf{msg}, \mathsf{salt}, h'_1, \left(H_1^{(\ell)}, \ldots, H_D^{(\ell)}\right)_{\ell\in[\tau]}\right)$
12 : Output accept if $h'_1 = h_1$ and $h'_2 = h_2$, otherwise output reject
```

Figure 8: Verification algorithm of MiRitH-Hypercube.

4 Security Evaluation

The security of MiRitH formally relies on the security of the ZKPoK from Section 3.1.2, the Fiat–Shamir transform, and the hardness of solving random instances of the MinRank problem.

The Fiat–Shamir transform is known to be secure in the Random Oracle Model (ROM) and has been the target of extensive security analysis since its invention.

The completeness, soundness, and zero-knowledge properties of the sigma protocol as well as the fact that the resulting signature scheme from Section 3 is existentially unforgeable under chosen messages attacks (EU-CMA) are proved in the ROM in Section 9.

The hardness of the MinRank problem is well established in cryptography, since, due to its many applications in cryptanalysis, algorithms to solve the problem have been extensively studied. In the following, we give an overview of the best-known attacks and their complexities.

4.1 Known attacks to MinRank

There are two types of attacks on the MinRank problem: Combinatorial and algebraic. On the combinatorial side, we have the well-known kernel-search attack. In contrast, on the algebraic side, several approaches model a MinRank instance as a system of multivariate polynomial equations [FEDS13, KS99, VBC$^+$19]. Still, they remain less efficient versions of the support-minors modeling

[BB22, GD22]. In the following, we detail the most efficient approaches of both categories, as well as a recent hybrid strategy.

In this section, $\omega \in [2,3]$ denotes the constant of matrix multiplication, i.e., two $n \times n$ matrices over a field can be multiplied in $\mathcal{O}(n^\omega)$ field operations.

4.1.1 Enumeration

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

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

Finally, we can enumerate the entries of the matrix K, which is equivalent to searching elements in the kernel of the hidden low-rank matrix. The following section explains this attack in a more general fashion.

4.1.2 Kernel-search

The kernel-search algorithm was introduced by Goubin and Courtois [GC00]. It consists of guessing $\lceil k/m \rceil$ linearly independent vectors in the kernel of the unknown rank r matrix E. The expected complexity of this algorithm, in terms of multiplications in $\mathbb{F}_q$, is

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

4.1.3 Support-minors modeling

The support-minors (SM) modeling is an algebraic method to solve the MinRank problem, which was introduced by Bardet et al. [BBC+20]. It models an instance of the MinRank problem as a bilinear system of equations that are then solved by an XL-like algorithm.

For $q > 2$, the complexity of solving the SM equations is computed as

$$\min\left\{3 \cdot k(r+1) \cdot A(b,n')^2, 7 \cdot A(b,n')^\omega \;\middle|\; \substack{1 \le b \le r+1,\\ r+b \le n' \le n,\\ A(b,n')-1 \le B(b,n')}\right\},$$

where

$$A(b,n') := \binom{n'}{r}\binom{k+b-1}{b},$$

$$B(b,n') := \sum_{j=1}^{b}\sum_{i=1}^{j}\left\{(-1)^{i+1}\binom{n'}{r+i}\binom{m+i-1}{i}\binom{k}{j-i}\right\}.$$

In Bardet et al. [BBB+22], the authors showed a new guess-and-solve (or hybrid) approach for the MinRank, which is more efficient than directly solving a given instance.

[3]Originally called the big-m attack, since m was the variable used to denote the number of matrices in the MinRank problem [Cou01].

4.1.4 Hybrid approaches

We consider two hybridization approaches to estimate the number of multiplications in $\mathbb{F}_q$ required to solve a given MinRank instance. The first approach guesses l_v coefficients of the solution vector $\boldsymbol{\alpha}$. The second approach, introduced by Bardet et al. [BBB+22], guesses a vectors (with a specific structure) in the right kernel of the secret E. When both approaches are combined, one derives MinRank instances $\tilde{\boldsymbol{M}}$ with parameters $(q, m, n-a, k-am-l_v, r)$. Hence the complexity of this hybrid approach is given by

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

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

Note that, while improving on state of the art, the impact on parameter selection for MiRitH by this attack has been moderate with decreasing the security level of our conservative instantiations by about 4 (category I) to 14 bits (category V).

4.1.5 Quantum attacks

There are at least two ways to do a Grover search in order to solve the MinRank problem. The first consists of searching the solution vector $\boldsymbol{\alpha} \in \mathbb{F}_q^k$. The other consists of searching some columns of the secret matrix K from Eq. (2).

For our proposed parameters (Table 1), we found that the quantum search over the entries of K is more efficient than the search over the coordinates of $\boldsymbol{\alpha}$. Hence, we focus on describing this attack.

Proposition 1. *Let $\boldsymbol{M} = (M_0, \ldots, M_k)$ be a MinRank instance, where $M_\ell = [\mu_{(i,j)}^{(\ell)}]_{i,j=1}^{m,n} \in \mathbb{F}_q^{m\times n}$ with solution $\boldsymbol{\alpha}$. Let $K = [\kappa_{(i,j)}]_{i,j=1}^{r,n-r} \in \mathbb{F}_q^{r\times(n-r)}$ be as in Eq. (2) and $1 \le t \le n-r$ be an integer. For $\mathbf{x}_t = (x_{(1,1)}, x_{(1,2)}, \ldots, x_{(r,t)}) \in \mathbb{F}_q^{rt}$ define*

$$B(\mathbf{x}_t) = \begin{pmatrix} B_1(\mathbf{x}_t) \\ \vdots \\ B_t(\mathbf{x}_t) \end{pmatrix}^{\top} \in \mathbb{F}_q^{(k+1)\times(mt)},$$

where

$$B_j(\mathbf{x}_t) = -\begin{pmatrix} \mu_{(1,j)}^{(1)} & \ldots & \mu_{(1,j)}^{(k)} & \mu_{(1,j)}^{(0)} \\ \vdots & & \vdots & \vdots \\ \mu_{(m,j)}^{(1)} & & \mu_{(m,j)}^{(k)} & \mu_{(m,j)}^{(0)} \end{pmatrix} +$$

$$+\begin{pmatrix} \sum_{i=1}^{r}\mu_{(1,n-r+i)}^{(1)}x_{(i,j)} & & \sum_{i=1}^{r}\mu_{(1,n-r+i)}^{(k)}x_{(i,j)} & \sum_{i=1}^{r}\mu_{(1,n-r+i)}^{(0)}x_{(i,j)} \\ \vdots & \ldots & \vdots & \vdots \\ \sum_{i=1}^{r}\mu_{(m,n-r+i)}^{(1)}x_{(i,j)} & & \sum_{i=1}^{r}\mu_{(m,n-r+i)}^{(k)}x_{(i,j)} & \sum_{i=1}^{r}\mu_{(m,n-r+i)}^{(0)}x_{(i,j)} \end{pmatrix}.$$

Then it holds that

$$(\alpha_1, \dots, \alpha_k, 1) \cdot B(\boldsymbol{\kappa}_t) = \mathbf{0}.$$

Note that Proposition 1 implies that the matrix $B(\boldsymbol{\kappa}_t)$ is not of full rank. For the attack, we ensure by choice of t that the only $\mathbf{x}_t$ that leads to a $B(\mathbf{x}_t)$ with a rank defect is $\boldsymbol{\kappa}_t$. Therefore, we fix $\lceil k/m \rceil \le t \le n - r$ such that $rt \le \binom{mt}{k+1}$. Hence $k + 1 \le mt$, and $\boldsymbol{\kappa}_t \in \mathbb{F}_q^{rt}$ is expected to be the only vector such that $B(\boldsymbol{\kappa}_t)$ is not full-rank. Indeed, the total number of different matrices $B(\boldsymbol{\gamma})$ with $\boldsymbol{\gamma} \in \mathbb{F}_q^{rt}$ is at most q^{rt}, each of them has $\binom{mt}{k+1}$ maximal minors, and any of these minors is zero with probability close to $1/q$. Finally, note that once $\boldsymbol{\kappa}_t$ is known, the MinRank solution $(\alpha_1, \dots, \alpha_k)$ can be efficiently recovered by classical Gaussian elimination using the relation from Proposition 1.

Define $\Psi : \left(\mathbb{F}_2^{\log q}\right)^{rt} \to \mathbb{F}_q^{rt}$ such that $\Psi(\mathbf{v}_1, \dots, \mathbf{v}_{rt}) := (\gamma_1, \dots, \gamma_{rt})$, where $\mathbf{v}_i$ is the binary representation of γ_i. We use Grover's algorithm to find the vector $\boldsymbol{\kappa}_t \in \mathbb{F}_q^{rt}$ representing the first t columns of the secret matrix K. This algorithm uses a quantum oracle $\mathcal{F} : \mathbb{F}_2^{rt \log q} \to \mathbb{F}_2$ such that

$$\mathcal{F}(\mathbf{v}) = \begin{cases} 1 & \text{if } \Psi(\mathbf{v}) = \boldsymbol{\kappa}_t \\ 0 & \text{otherwise.} \end{cases}$$

Our quantum oracle $\mathcal{F}$ on input $\mathbf{v} \in \mathbb{F}_2^{rt \log q}$ splits into three parts:

1. Compute the matrix $B(\boldsymbol{\gamma})$, where $\boldsymbol{\gamma} = \Psi(\mathbf{v})$.
2. Compute, by Gaussian elimination, the row-reduced form of $B(\boldsymbol{\gamma})$.
3. Negate every qubit representing the last column of $B(\boldsymbol{\gamma})$, and then output the multiplication of such qubits.

Notice $\mathcal{F}(\mathbf{v}) = 1$ if and only if at the end of step 2, the last row of $B(\boldsymbol{\gamma})$ is the zero vector.

Grover's algorithm states that the vector $\boldsymbol{\kappa}_t$ can be found wit high probability after $\mathcal{O}\big(2^{(rt \log q)/2}\big)$ calls to the quantum oracle $\mathcal{F}$. The complexity of computing $\mathcal{F}$ is dominated by the Gaussian elimination step, which can be performed in $\mathcal{O}\left((k+1)^2 \cdot mt\right)$ operations in $\mathbb{F}_q$. Therefore, we estimate our quantum complexity to be

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

operations over $\mathbb{F}_q$.

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

We lower bound the depth of the described quantum circuit by

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

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

In the case of $D > 2^{\texttt{maxdepth}}$, the most efficient strategy to restrict the depth of the quantum circuit is to partition the search space in P equally sized sets [Zal99]. Then the search has to be reapplied for each partition, which comes at a depth of

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

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

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

$$T = \mathcal{O}(P \cdot D_P \cdot mt \log^2 q), \tag{5}$$

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

4.2 Known attacks to the protocol

It is possible to forge a signature without solving the underlying instance of the MinRank problem. For signature schemes built by applying the Fiat-Shamir transformation on a five-pass identification scheme, the best-known attack of this type was introduced by Kales and Zaverucha [KZ20]. It is known as the KZ attack.

In our case, the complexity of the KZ attack is measured as the expected number of calls to Hash_1 and Hash_2. And, it is given by

$$C(\tau, q, s, N) = \min_{0 \le k \le \tau} \left\{ \frac{1}{\sum_{i=k}^{\tau} \left(\frac{1}{q^s}\right)^i \left(1 - \frac{1}{q^s}\right)^{\tau - i} \binom{\tau}{i}} + N^{\tau - k} \right\}. \tag{6}$$

5 Parameters

In Table 1 and Table 2, we propose parameter sets for MiRitH and MiRitH-Hypercube, respectively, for the different security levels I, III, and V defined by NIST, which correspond to 143, 207, and 272 bits of classical security, respectively [NIS].

The MinRank problem parameters (q, m, n, k, r) are chosen with respect to the best-known attacks sketched in Section 4.1. In Table 3, we state the estimated quantum security margin for each parameter set. That is the quotient of the gates necessary to launch a quantum attack according to Eq. (5) and the defined security threshold of $2^{b-\texttt{maxdepth}}$ for $b = 157, 221, 285$ for category I, III and V respectively. Note that all parameter sets have a positive margin, i.e., it offers a higher quantum security than AES with corresponding keylength.

The number of rows of the challenge matrices R, the number of parties, N for MiRitH and N^D for MiRitH-Hypercube, in the underlying MPC protocol and the number τ of repetitions of the identification protocol are computed such that the forgery cost in Eq. (6) is at least 2^λ. We provide two classes of parameter sets, namely: *a* and *b* sets. The *Xa* sets minimize the signature size, while precisely matching the NIST security levels definitions. On the other hand, with the *Xb* sets, we provide a more conservative choice of parameters which include some security margin to account for potential future attack improvements.

Further, for each parameter set, we provide a "Fast" and a "Short" (a "Shorter" and a "Shortest", in addition, for MiRitH-Hypercube) variant leveraging a trade-off between signing/verification time and signature size that originates from the use of the MPC protocol. In fact, lowering the number of MPC parties allows one to reduce the computation time, but to maintain the same forgery cost, the number of repetitions has to be increased, which in turn results in larger signature sizes.

Set	Variant	λ	q	m	n	k	r	s	N	τ	Bit security
Ia	fast	128	16	15	15	78	6	5	16	39	144
	short							9	256	19	
Ib	fast	128	16	16	16	142	4	5	16	39	159
	short							9	256	19	
IIIa	fast	192	16	19	19	109	8	7	16	55	207
	short							9	256	29	
IIIb	fast	192	16	19	19	167	6	7	16	55	232
	short							9	256	29	
Va	fast	256	16	21	21	189	7	7	16	74	273
	short							10	256	38	
Vb	fast	256	16	22	22	254	6	7	16	74	301
	short							10	256	38	

Table 1: Parameters of MiRitH.

Set	Variant	λ	q	m	n	k	r	s	N	D	τ	Bit security
Ia	fast	128	16	15	15	78	6	5	2	4	39	144
	short							9	2	8	19	
	shorter							12	2	12	13	
	shortest							12	2	16	10	
Ib	fast	128	16	16	16	142	4	5	2	4	39	159
	short							9	2	8	19	
	shorter							12	2	12	13	
	shortest							12	2	16	10	
IIIa	fast	192	16	19	19	109	8	7	2	4	55	207
	short							9	2	8	29	
	shorter							13	2	12	19	
	shortest							13	2	16	15	
IIIb	fast	192	16	19	19	167	6	7	2	4	55	232
	short							9	2	8	29	
	shorter							13	2	12	19	
	shortest							13	2	16	15	
Va	fast	256	16	21	21	189	7	10	2	4	71	273
	short							10	2	8	38	
	shorter							14	2	12	26	
	shortest							14	2	16	20	
Vb	fast	256	16	22	22	254	6	10	2	4	71	301
	short							10	2	8	38	
	shorter							14	2	12	26	
	shortest							14	2	16	20	

Table 2: Parameters of MiRitH-Hypercube.

Set	t	log(gates)
Ia	6	23
Ib	10	43
IIIa	6	9
IIIb	9	36
Va	10	37
Vb	12	47

Table 3: Quantum security margin of MiRitH and MiRitH-Hypercube

6 Implementation Notes

6.1 Linear algebra

6.1.1 Field elements

We proposed parameters for our scheme over the field $\mathbb{F}_{16}$, in which every element is represented by a binary polynomial of degree 3, e.g., $a_3x^3 + a_2x^2 + a_1x + a_0$.

In our implementation, the multiplication of two elements in $\mathbb{F}_{16}$ is implemented as a polynomial multiplication modulus the polynomial $x^4 + x + 1$. An element $a_3x^3 + a_2x^2 + a_1x + a_0 \in \mathbb{F}_{16}$ is stored as $a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2 + a_0$.

6.1.2 Matrices

A matrix M of dimensions $n_r \times n_c$ is represented as a byte-array $\mathtt{A}_M$ of length $\lfloor (n_r + 1)/2 \rfloor n_c$. The array $\mathtt{A}_M$ is the concatenation of n_c byte-arrays $\mathtt{A}_{M,0}, \mathtt{A}_{M,1}, \dots, \mathtt{A}_{M,n_c-1}$, where the array $\mathtt{A}_{M,j}$ stores the j-th column of M.

Let $\epsilon_{i,j}$ be the (i, j) entry of M. Then, for $0 \leq j < n_c$ and $0 \leq i < \lfloor (n_r - 2)/2 \rfloor$, $\mathtt{A}_{M,j}[i] = 16 \cdot \epsilon_{2i+1,j} + \epsilon_{2i,j}$. If n_r is odd, $\mathtt{A}_{M,j}[(n_r + 1)/2 - 1] = \epsilon_{n_r-1,j}$.

6.2 Data packing

Our scheme employs five types of data, namely: seeds, salts, hash digests, (non-negative) integers, and matrices. Seeds, salts, hash digests, and integers are packed in the natural way and have sizes of λ bits, 2λ bits, 2λ bits, and 32 bits, respectively. The packing of matrices is explained in the next section. Hereafter, for each object A of the aforementioned five data types, we write $\mathsf{pack}(A)$ for the string of bits of the packing of A.

6.2.1 Matrices

We pack a matrix $M \in \mathbb{F}_{16}^{n_r \times n_c}$ into a string of byte $\mathtt{P}_M = \mathsf{pack}(M)$ in two different ways, which differ on how we use the first byte $\mathtt{P}_M[0]$ of the byte string $\mathtt{P}_M$ to store elements of M.

We use a binary parameter, called b_o, to indicate how the first byte of $\mathtt{P}_M$ should be used to pack entries of M. When $b_o = 0$, we use $\mathtt{P}_M[0]$ to pack two entries of M. Otherwise, when $b_o = 1$, we pack into $\mathtt{P}_M[0]$ only one entry of M.

The reason for the matrix packing strategy when $b_o = 1$ is that, while packing a signature, we have to pack several matrices, and some of them might have an odd number of entries. Hence, we cannot fill a whole byte array $\mathtt{P}_M$ only with the entries of M; at least one byte will be half-filled. Such a half-filled byte of $\mathtt{P}_M$ must (otherwise, we could trivially forge a signature) be then used to pack one entry of another matrix during the signature packing process.

Now we precisely describe how the matrix packing works in our reference implementation. In what follows, for $i = 0, \dots, n_r n_c - 1$, we denote by e_i the ith entry of M in a column-major order:

- If n_r is even:
  - If $b_o = 0$, we set
  $$\mathtt{P}_M[i] = 16 \cdot e_{2i+1} + e_{2i}, \text{ for } i = 0, \dots, (n_r n_c - 2)/2.$$
  - If $b_o = 1$, we set $\mathtt{P}_M[0] = 16 \cdot e_{n_r n_c - 1}$
  $$\mathtt{P}_M[i] = 16 \cdot e_{2i-1} + e_{2i-2}, \text{ for } i = 1, \dots, (n_r n_c - 2)/2,$$
  and $\mathtt{P}_M[(n_r n_c - 2)/2 + 1] = e_{n_r n_c - 2}$.
- If n_r odd: Let R_{n_r} denote the last row of M, $\gamma_0, \dots, \gamma_{n_c - 1}$ the entries of R_{n_r}, and let $M_{sub} \in \mathbb{F}_{16}^{(n_r - 1) \times n_c}$ be the matrix that results from removing R_{n_r} from M.
  - If $b_o = 0$, we first pack M_{sub} as in the n_r-even case. Then, we set
  $$\mathtt{P}_M\left[\frac{(n_r - 1)n_c}{2} + i\right] = 16 \cdot \gamma_{2i+1} + \gamma_{2i}, \text{ for } i = 0, \dots, \lfloor (n_c - 2)/2 \rfloor.$$
  If n_c is odd, then we set $\mathtt{P}_M[(n_r n_c - 1)/2] = \gamma_{n_c - 1}$.
  - If $b_o = 1$, we first set $\mathtt{P}_M[0] = 16 \cdot \gamma_{n_c - 1}$. Then, we pack M_{sub} as in the n_r-even case with $b_o = 0$ starting from $\mathtt{P}_M[1]$. Finally, we set
  $$\mathtt{P}_M\left[\frac{(n_r - 1)n_c}{2} + 1 + i\right] = 16 \cdot \gamma_{2i+1} + \gamma_{2i}, \text{ for } i = 0, \dots, \lfloor (n_c - 3)/2 \rfloor.$$
  If n_c is even, then we set $\mathtt{P}_M[(n_r n_c)/2 - 1] = \gamma_{n_c - 2}$.

6.2.2 Public key

The compressed public key is stored as $\mathsf{pack}(\mathsf{seed}_{\mathsf{pk}}) \| \mathsf{pack}(M_0)$, where $\mathsf{seed}_{\mathsf{pk}}$ is the seed to generate the matrices $M_1, \dots, M_k$ (see Fig. 4).

6.2.3 Secret key

The compressed secret key consists only of the seed $\mathsf{seed}_{\mathsf{sk}}$ to generate the vector $\boldsymbol{\alpha}$ (see Fig. 4). Hence, it can be stored as $\mathsf{pack}(\mathsf{seed}_{\mathsf{sk}})$.

However, in order to sign a message, our schemes require both the public key and the secret key, while SUPERCOP signing function crypto_sign allows one to pass as an argument only the secret key. Therefore, to make our implementations compatible with SUPERCOP, we have to store into the secret key also a copy of the public key. Thus, the secret key is stored as $\mathsf{pack}(\mathsf{seed}_{\mathsf{sk}})\|\mathsf{pack}(\mathsf{seed}_{\mathsf{pk}})\|\mathsf{pack}(M_0)$.

6.2.4 Signature

The signature of MiRitH is packed as described in Fig. 9. The signature of MiRitH-Hypercube is packed as described in Fig. 9 but with N replaced by N^D.

```
PackSignature($\sigma$)
1 :  salt, $h_1, h_2, \Big(\big(\mathsf{state}_i^{(\ell)}\big)_{i \neq i^{*,(\ell)}}, \mathsf{com}_{i^{*,(\ell)}}^{(\ell)}, [\![S^{(\ell)}]\!]_{i^{*,(\ell)}}\Big)_{\ell \in [\tau]} \leftarrow \sigma$
2 :  $\mathsf{packed}_{\mathsf{sgn}} \leftarrow \mathsf{salt}\|h_1\|h_2$
3 :  for $\ell \in [\tau]$ do
4 :    $\mathsf{packed}_{\mathsf{sgn}} \leftarrow \mathsf{packed}_{\mathsf{sgn}}\|\mathsf{pack}(\mathsf{com}_{i^{*,(\ell)}}^{(\ell)})$
       $\mathsf{packed}_{\mathsf{sgn}} \leftarrow \mathsf{packed}_{\mathsf{sgn}}\|\mathsf{Tree\_pack}(\mathsf{tree}^{(\ell)}, i^{*,(\ell)}, N)$
5 :    // $\mathsf{tree}^{(\ell)}$ is the tree of seeds of the $\ell$th round
6 :  // Matrices are packed last because their sizes are not always
7 :  // multiples of 8 bits, and so one has to take care of bit offsets.
8 :  for $\ell \in [\tau]$ do
9 :    if $i^{*,(\ell)} \neq N$ then
10 :     $\mathsf{packed}_{\mathsf{sgn}} \leftarrow \mathsf{packed}_{\mathsf{sgn}}\|\mathsf{pack}([\![\boldsymbol{\alpha}^{(\ell)}]\!]_N)\|\mathsf{pack}([\![K^{(\ell)}]\!]_N)\|\mathsf{pack}([\![C^{(\ell)}]\!]_N)$
11 :    $\mathsf{packed}_{\mathsf{sgn}} \leftarrow \mathsf{packed}_{\mathsf{sgn}}\|\mathsf{pack}([\![S]\!]_{i^{*,(\ell)}}^{(\ell)})$
     return $\mathsf{packed}_{\mathsf{sgn}}$
```

Figure 9: Algorithm for the signature packing of MiRitH.

6.3 Pseudorandom generator

To implement our pseudorandom generator (PRG), we employed the SHAKE256 extendable-output functions (XOF) [Div14]. The pseudocode of PRG is shown in Fig. 10. It consists of the initialization function PRG_init, which returns the state of a new PRG initialized by salt (2λ bits) and seed (λ bits), and the function PRG_bytes, which returns the updated state of the PRG and a string of num_bytes pseudorandom bytes. In Fig. 10, SHAKE256_init() is a function that returns a new instance of the shake object, SHAKE256_absorb(shake_inst, string) is a function that returns the updated shake instance shake_inst after absorbing the string of bits string, and

```
PRG_init(salt, seed)
1 :  entropy ← $\emptyset$
2 :  if salt $\neq 0$ then
3 :    entropy ← entropy||salt
4 :  if seed $\neq 0$ then
5 :    entropy ← entropy||seed
6 :  prg_state ← SHAKE256_init()
7 :  prg_state ← SHAKE256_absorb(prg_state, entropy)
     return prg_state
```

```
PRG_bytes(prg_state, num_bytes)
return SHAKE256_squeeze(prg_state, num_bytes)
```

Figure 10: Functions to initialize the PRG and to get random bytes from the PRG.

SHAKE256_squeeze(shake_inst, num_bytes) is a function that returns the updated shake instance shake_inst and a string of num_bytes squeezed from shake_inst.

In the next two sections, Section 6.3.1 and Section 6.3.2, we provide details on the generation of pseudorandom matrices and pseudorandom integers in terms of PRG_init and PRG_bytes.

6.3.1 Pseudorandom matrices

We generate pseudorandom matrices by using the algorithm of Fig. 11. The function PRG_matrix returns the updated prg_state and a pseudorandom $n_r \times n_c$ matrix.

```
PRG_matrix(prg_state, $n_r$, $n_c$)
1 :  matrix ← $\emptyset$
2 :  $h \leftarrow \lfloor n_r/2 \rfloor + (n_r \bmod 2)$
3 :  for $j \in [n_c]$ do
4 :    prg_state, $b_1, b_2, \cdots, b_h$ ← PRG_bytes(prg_state, $h$)
5 :    if $n_r \bmod 2 = 1$ then
6 :      $b_h \leftarrow b_h \otimes 15$   // $\otimes$ means bitwise multiplication
7 :    matrix ← matrix||$b_1$||$b_2$||$\cdots$||$b_h$
     return prg_state, matrix
```

Figure 11: Algorithm for the generation of a pseudorandom matrix.

In particular, the pseudorandom matrices of the first challenge are generated as described in Fig. 15. We use the function PRG_matricesPK (Fig. 12) to generate the matrices $M_1, \ldots, M_k$ in the public key, while we use the function PRG_matricesSK (Fig. 13) to generate the secret matrices $\boldsymbol{\alpha}$, K

and E^R. The algorithm PRG_shares (Fig. 14) is used to generate the matrix shares $[\![A]\!]_i$, $[\![\boldsymbol{\alpha}]\!]_i$, $[\![K]\!]_i$ and $[\![C]\!]_i$.

```
PRG_matricesPK(seed)
1 :  prg_state ← PRG_init(NULL, seed)
2 :  for $i \in [k]$ do
3 :    prg_state, $M_i$ ← PRG_matrix(prg_state, $m, n$)
     return $M_1, \dots, M_k$
```

Figure 12: Algorithm for the generation of the public matrices $M_1, \dots, M_k \in \mathbb{F}_q^{m \times n}$.

```
PRG_matricesSK(seed)
1 :  prg_state ← PRG_init(NULL, seed)
2 :  prg_state, $\boldsymbol{\alpha}$ ← PRG_matrix(prg_state, $k, 1$)
3 :  prg_state, $K$ ← PRG_matrix(prg_state, $r, n-r$)
4 :  prg_state, $E^R$ ← PRG_matrix(prg_state, $m, r$)
     return $\boldsymbol{\alpha}, K, E^R$
```

Figure 13: Algorithm for the generation of the secret vector $\boldsymbol{\alpha} \in \mathbb{F}_q^{k \times 1}$ and the secret matrices $K \in \mathbb{F}_q^{r \times (n-r)}$ and $E^R \in \mathbb{F}_q^{m \times r}$.

```
PRG_shares(salt, seed, $i$, $\tilde{N}$)
1 :  prg_state ← PRG_init(salt, seed)
2 :  if $i \neq \tilde{N}$ then
3 :    prg_state, $[\![A]\!]_i$ ← PRG_matrix(prg_state, $s, r$)
4 :    prg_state, $[\![\boldsymbol{\alpha}]\!]_i$ ← PRG_matrix(prg_state, $k, 1$)
5 :    prg_state, $[\![K]\!]_i$ ← PRG_matrix(prg_state, $r, n-r$)
6 :    prg_state, $[\![K]\!]_i$ ← PRG_matrix(prg_state, $s, n-r$)
7 :    return $[\![A]\!]_i, [\![\boldsymbol{\alpha}]\!]_i, [\![K]\!]_i, [\![C]\!]_i$
8 :  else
9 :    prg_state, $[\![A]\!]_{\tilde{N}}$ ← PRG_matrix(prg_state, $s, r$)
       return $[\![A]\!]_{\tilde{N}}$
```

Figure 14: Algorithm for the generation of the pseudorandom shares $[\![A]\!]_i \in \mathbb{F}_q^{s \times r}$, $[\![\boldsymbol{\alpha}]\!]_i \in \mathbb{F}_q^{k \times 1}$, $[\![K]\!]_i \in \mathbb{F}_q^{r \times (n-r)}$ and $[\![C]\!]_i \in \mathbb{F}_q^{s \times (n-r)}$.

```
PRG_first_challenge($h_1$)
1 :  prg_state ← PRG_init($h_1$, NULL)
2 :  for $\ell \in [\tau]$ do
3 :    prg_state, $R_\ell$ ← PRG_matrix(prg_state, $s, m$)
     return $R_1, R_2, \ldots, R_\tau$
```

Figure 15: Algorithm for the generation of the first challenge.

6.3.2 Pseudorandom integers

A pseudorandom integer in an interval $[0, a)$ is produced by generating a pseudorandom 32-bits integer, and then reducing it modulo a.

In particular, the second challenge is generated as described in Fig. 16.

```
PRG_second_challenge($h_2, \tilde{N}$)
1 :  prg_state ← PRG_init($h_2$, NULL)
2 :  prg_state, $b_1, b_2, \cdots, b_{4\tau}$ ← PRG_bytes(prg_state, $4\tau$)
3 :  for $\ell \in [\tau]$ do
4 :    $i^{*,(\ell)} \leftarrow$ uint($b_{4\ell-3} \| b_{4\ell-2} \| b_{4\ell-1} \| b_{4\ell}$) mod $\tilde{N}$
       // uint($x$) denotes the non-negative integer having binary representation $x$.
     return $i^{*,(1)}, i^{*,(2)}, \ldots, i^{*,(\tau)}$
```

Figure 16: Algorithm for the generation of the second challenge.

6.4 Seed trees

Fig. 17 shows a pseudocode for TreePRG, which is the function used to generate the seeds of the parties for every round. The inputs salt and seed are byte strings of size 2λ and λ, respectively. The string tree is a sequence of byte strings of size λ, where tree$[i]$ denotes the ith string in tree from left to right.

```
TreePRG(salt, seed, $\tilde{N}$)
1 : tree ← seed
2 : for $i \in \left[2^{\lceil \log_2 \tilde{N} \rceil} - 1\right]$ do
3 :     prg_state ← PRG_init(salt, tree[$i$])
4 :     // tree[$i$] denotes the $i$th seed appended to tree
5 :     prg_state, child_seed$_l$ ← PRG_bytes(prg_state, $\lambda$)
6 :     prg_state, child_seed$_r$ ← PRG_bytes(prg_state, $\lambda$)
7 :     tree ← tree||child_seed$_l$||child_seed$_r$
8 : $j \leftarrow 2^{\lceil \log_2 \tilde{N} \rceil} - 1$
9 : (seed$_i$)$_{i \in [\tilde{N}]}$ ← (tree[$i$])$_{i=j+1,\dots,j+\tilde{N}}$
    return (seed$_i$)$_{i \in [\tilde{N}]}$
```

Figure 17: The TreePRG function.

In Fig. 18 we show the pseudocode to pack and unpack the seeds used by all the parties but the i^*th in a particular round. The function Tree_pack takes as input a sequence, called tree, of $2^{\lceil \log_2 N \rceil + 1} - 1$ seeds and two integers i^*, N with $1 \leq i^* \leq N$. It outputs a sequence of seeds packed of length $\lceil \log_2 N \rceil$. The function Tree_unpack receives as input a sequence packed as the output of Tree_pack and two integers i^*, N. It outputs a sequence of length N, where in every entry $i \in [N] \setminus \{i^*\}$ it has a seed, but it contains $\emptyset$ in the i^*th entry, i.e., it is empty in the i^*th entry.

```
$\mathsf{Tree\_pack}(\mathsf{tree}, i^*, \tilde{N})$
1 :  $\mathsf{idx} \leftarrow 1$
2 :  $\mathsf{packed} \leftarrow \emptyset$
3 :  for $j \in \left[\lceil \log_2 \tilde{N} \rceil\right]$ do
4 :      $t \leftarrow \left\lfloor \frac{i^*}{2^{\lceil \log_2 \tilde{N} \rceil - j}} \right\rfloor$
5 :      // $\mathsf{tree}$ is the concatenation several seeds.
6 :      // $\mathsf{tree}[i]$ denotes the $i$th seed in $\mathsf{tree}$.
7 :      if $t \bmod 2 = 1$ then
8 :          $\mathsf{packed} \leftarrow \mathsf{packed} \| \mathsf{tree}[2 \cdot \mathsf{idx}]$
9 :          $\mathsf{idx} \leftarrow 2 \cdot \mathsf{idx} + 1$
10 :     else
11 :         $\mathsf{packed} \leftarrow \mathsf{packed} \| \mathsf{tree}[2 \cdot \mathsf{idx} + 1]$
12 :         $\mathsf{idx} \leftarrow 2 \cdot \mathsf{idx}$
     return $\mathsf{packed}$
```

```
$\mathsf{Tree\_unpack}(\mathsf{packed}, i^*, \tilde{N})$
1 :  $\mathsf{temp} \leftarrow (\emptyset)_{i \in \left[2^{\lceil \log_2 \tilde{N} \rceil}\right]}$
2 :      // $\mathsf{temp}$ is an empty sequence of length $2^{\lceil \log_2 \tilde{N} \rceil}$.
3 :      // $\mathsf{temp}[i]$ denotes the $i$th coordinate of $\mathsf{temp}$.
4 :  $j \leftarrow \lceil \log_2 \tilde{N} \rceil - 1$
5 :  $\mathsf{gap} \leftarrow 0$
6 :      // $\mathsf{packed}$ is the concatenation several seeds.
7 :      // $\mathsf{packed}[i]$ denotes the $i$th seed in $\mathsf{packed}$.
8 :  while $j > -1$ do
9 :      $j_0 \leftarrow \left\lfloor \frac{i^*}{2^j} \right\rfloor \bmod 2$
10 :     $\mathsf{aux}_{\mathsf{idx}} \leftarrow (1 - j_0) \cdot 2^j$
11 :     $\mathsf{idx} \leftarrow \mathsf{aux}_{\mathsf{idx}} + \mathsf{gap}$
12 :     $\mathsf{gap} \leftarrow \mathsf{gap} + j_0 \cdot 2^j$
13 :     $\mathsf{tree}_0 \leftarrow \mathsf{TreePRG}(\mathsf{salt}, \mathsf{packed}[\lceil \log_2 \tilde{N} \rceil - j], 2^j)$
14 :     for $i \in [2^j]$ do
15 :         $\mathsf{temp}[\mathsf{idx} + i] \leftarrow \mathsf{tree}_0[i]$
16 :  $\mathsf{unpacked} \leftarrow (\mathsf{temp}[i])_{i \in [\tilde{N}]}$
     return $\mathsf{unpacked}$
```

Figure 18: Packing and unpacking of seeds.

6.5 Hash functions

Our hash functions Hash_0, Hash_1, Hash_2, $\mathrm{Hash}_{(1,0)}$, $\mathrm{Hash}_{(1,1)}$, $\mathrm{Hash}_{(2,0)}$ and $\mathrm{Hash}_{(2,1)}$ are implemented by using $\mathsf{SHA3}$ [Div14]. Specifically, for $\lambda = 128, 192, 256$ we define $\mathsf{SHA3}$ as $\mathsf{SHA3}$-256, $\mathsf{SHA3}$-384, $\mathsf{SHA3}$-512, respectively.

The hash functions used in MiRitH and MiRitH-Hypercube are defined as follows:

- $\mathrm{Hash}_0\left(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)}\right) := \mathsf{SHA3}\left(\mathsf{salt} \| \mathsf{pack}(\ell) \| \mathsf{pack}(i) \| \mathsf{pack}(\mathsf{state}_i^{(\ell)})\right)$, where

  $$\mathsf{pack}(\mathsf{state}_i^{(\ell)}) := \begin{cases} \mathsf{state}_i^{(\ell)} & \text{if } i \neq N \\ \mathsf{seed}_N^{(\ell)} \| \mathsf{pack}(\llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_N) \| \mathsf{pack}(\llbracket K^{(\ell)} \rrbracket_N) \| \mathsf{pack}(\llbracket C^{(\ell)} \rrbracket_N) & \text{otherwise} \end{cases},$$

  and $\mathsf{state}_N^{(\ell)} = (\mathsf{seed}_N^{(\ell)}, \llbracket \boldsymbol{\alpha}^{(\ell)} \rrbracket_N, \llbracket K^{(\ell)} \rrbracket_N, \llbracket C^{(\ell)} \rrbracket_N)$.

- $\mathrm{Hash}_1\left(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}_i^{(\ell)})_{i \in [N], \ell \in [\tau]}\right) := \mathsf{SHA3}\left(\mathsf{msg} \| \mathsf{salt} \| \mathsf{packed}_{\mathsf{com}}\right)$, where

```
$\mathsf{packed}_{\mathsf{com}} \leftarrow \emptyset$.
for $\ell \in [\tau]$ do
    $\mathsf{packed}_{\mathsf{com}} \leftarrow \mathsf{packed}_{\mathsf{com}} \| \mathsf{com}_1^{(\ell)} \| \mathsf{com}_2^{(\ell)} \| \cdots \| \mathsf{com}_N^{(\ell)}$
```

- $\mathrm{Hash}_2\big(\mathsf{msg}, \mathsf{salt}, h_1, \big([\![S^{(\ell)}]\!]_i, [\![V^{(\ell)}]\!]_i \big)_{i \in [N],\, \ell \in [\tau]}\big) \;:=\; \mathsf{SHA3}\big(\mathsf{msg} \| \mathsf{salt} \| h_1 \| \mathsf{packed}_{SV}\big)$, with packed_{SV} is computed as

```
$\mathsf{packed}_{SV} \leftarrow \emptyset$.
for $\ell \in [\tau]$ do
  for $i \in [N]$ do
    $\mathsf{packed}_{SV} \leftarrow \mathsf{packed}_{SV} \| \mathsf{pack}\left([\![S^{(\ell)}]\!]_i\right) \| \mathsf{pack}\left([\![V^{(\ell)}]\!]_i\right)$
```

- $\mathrm{Hash}_{(1,0)}\big(\mathsf{salt}, \ell, (\mathsf{com}_i^{(\ell)})_{i \in [N^D]}\big) := \mathsf{SHA3}\big(\mathsf{msg} \| \mathsf{pack}(\ell) \| \mathsf{com}_1^{(\ell)} \| \mathsf{com}_2^{(\ell)} \| \cdots \| \mathsf{com}_{N^D}^{(\ell)}\big)$
- $\mathrm{Hash}_{(1,1)}\big(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}^{(\ell)})_{\ell \in [\tau]}\big) := \mathsf{SHA3}(\mathsf{msg} \| \mathsf{salt} \| \mathsf{com}^{(1)} \| \mathsf{com}^{(2)} \| \cdots \| \mathsf{com}^{(\tau)})$
- $\mathrm{Hash}_{(2,0)}\big(\mathsf{salt}, \ell, \big([\![S^{(\ell)}]\!]_{(k,j)}, [\![V^{(\ell)}]\!]_{(k,j)}\big)_{j \in [N]}\big) : \mathsf{SHA3}\big(\mathsf{salt} \| \mathsf{pack}(\ell) \| \mathsf{packed}_{SV}\big)$, where

```
$\mathsf{packed}_{SV} \leftarrow \emptyset$.
for $j \in [N]$ do
  $\mathsf{packed}_{SV} \leftarrow \mathsf{packed}_{SV} \| \mathsf{pack}\left([\![S^{(\ell)}]\!]_{(k,j)}\right) \| \mathsf{pack}\left([\![V^{(\ell)}]\!]_{(k,j)}\right)$
```

- $\mathrm{Hash}_{(2,1)}\big(\mathsf{msg}, \mathsf{salt}, h_1, \big(H_1^{(\ell)}, \ldots, H_D^{(\ell)}\big)_{\ell \in [\tau]}\big) := \mathsf{SHA3}\big(\mathsf{msg} \| \mathsf{salt} \| h_1 \| \mathsf{packed}_H\big)$, where

```
$\mathsf{packed}_H \leftarrow \emptyset$.
for $\ell \in [\tau]$ do
  $\mathsf{packed}_H \leftarrow \mathsf{packed}_K \| H_1^{(\ell)} \| H_2^{(\ell)} \| \cdots \| H_D^{(\ell)}$
```

6.6 Implementations

We provide three implementations of both MiRitH and MiRitH-Hypercube, namely: one reference implementation and two optimized implementations. The first optimized implementation is for Intel® processors and optimizes the matrix arithmetic by using AVX2 instructions. The second optimized implementation is for ARM processors and optimizes the matrix arithmetic by using NEON instructions.

6.6.1 Reference implementation

This implementation is written entirely in ANSI C. It has the only purpose of showing how the proposed scheme can be implemented without employing any particular optimization.

6.6.2 AVX implementation

Our optimized implementation uses the Advanced Vector Extension (AVX) and Advanced Vector Extension 2 (AVX2) instruction sets, which are part of every X86 CPU since Intel Haswell generation from 2013. These instruction sets introduced 256-bits vector registers while following the Single-Instruction-Multiple-Data (SIMD) approach. Thus a broad variety of instructions are available, which allow one to manipulate a 256-bit vector as 8-/16-/32-/64-bits vector elements, while computing all vector elements at the same time in constant time.

The core idea of our implementation is the usage of the `vpblendvb` instruction. Given three registers `ymm_a`, `ymm_b`, `ymm_c` $\in \mathbb{F}_{2^8}^{32}$ this instruction returns each vector element by selecting either the corresponding vector element from `ymm_b` or `ymm_a` if the highest bit in the vector element of `ymm_c` is set or not. Given two elements $a = a_3x^3 + a_2x^2 + a_1x + a_0, b = b_3x^3 + b_2x^2 + b_1x + b_0 \in \mathbb{F}_{2^4}$, the multiplication $a \cdot b$ can be expressed as $\sum_{i=0}^{3} b_i x^i \cdot a$. Note that $b_i \in \mathbb{F}_2$, thus an addend is selected depending on b_i, which matches the usage of the `vpblendvb` instruction. The subsequent reduction is implemented via a small lookup table, which fits into three AVX 256-bit registers utilizing the `vpshufb` instruction.

6.6.3 NEON implementation

ARM introduced its SIMD solution in 2004 with the ARMv7 Instruction Set Architecture (ISA), which was extended for ARMv8 to process 128-bit registers. The core of our implementation is based on [BCH+23] by utilizing the `vmulq_p8` instruction. This instruction computes an 8-bits polynomial multiplication on 16 vector elements from two input registers. The subsequent reduction is an implementation using a lookup table.

7 Performance Analysis

7.1 Public key and secret key sizes

The public key and secret key sizes for the different parameter sets in Section 5 are the same for both MiRitH (Table 4) and MiRitH-Hypercube (Table 8). As given in Section 3.2, the public key size is $\lambda + mn \log q$ bits and the secret key is a λ-bit seed, from which the secret MinRank solution is derived.

7.2 MiRitH

7.2.1 Signature sizes

From Section 3.3, it follows that the maximum size (in bits) for a MiRitH signature is equal to

$$\underbrace{6\lambda}_{\mathsf{salt}, h_1, h_2} + \tau \left(\underbrace{(k + r(n-r) + s(n-r) + sr) \cdot \log q}_{[\![\boldsymbol{\alpha}^{(\ell)}]\!]_N, [\![K^{(\ell)}]\!]_N, [\![C^{(\ell)}]\!]_N, [\![S^{(\ell)}]\!]_{i^*,(\ell)}} + \underbrace{\lambda \cdot \log_2 N}_{(\mathsf{seed}_i^{(\ell)})_{i \neq i^*}} + \underbrace{2\lambda}_{\mathsf{com}_{i^*,(\ell)}} \right),$$

while the average size (in bits) is equal to

$$\underbrace{6\lambda}_{\mathsf{salt},h_1,h_2} + \tau\left(\underbrace{\frac{N-1}{N}}_{\Pr[i^{*,(\ell)}\neq N]} \cdot \underbrace{\big(k + r(n-r) + s(n-r)\big)\log q}_{[\![\boldsymbol{\alpha}^{(\ell)}]\!]_N, [\![K^{(\ell)}]\!]_N, [\![C^{(\ell)}]\!]_N} + \underbrace{sr\log q}_{[\![S^{(\ell)}]\!]_{i^*}} + \underbrace{\lambda\cdot\log N}_{(\mathsf{seed}_i^{(\ell)})_{i\neq i^{*,(\ell)}}} + \underbrace{2\lambda}_{\mathsf{com}_{i^{*,(\ell)}}}\right).$$

where s is the MPC protocol Π_s parameter, N the number of parties in one MPC protocol and τ the number of parallel repetitions, respectively. The values N and τ are set to achieve a soundness error smaller than $2^{-\lambda}$.

Table 4 shows the sizes of the public key, secret key, maximum signature and average signature for all the proposed parameter sets for MiRitH.

Set	Variant	Public key	Secret key	Signature (max)	Signature (avg)
Ia	fast	129	16	7,877	7,661
	short			5,673	5,665
Ib	fast	144	16	9,105	8,800
	short			6,309	6,298
IIIa	fast	205	24	17,139	16,668
	short			12,440	12,423
IIIb	fast	205	24	18,459	17,882
	short			13,136	13,115
Va	fast	253	32	30,458	29,568
	short			21,795	21,763
Vb	fast	274	32	33,048	31,980
	short			23,182	23,144

Table 4: Public key, secret key, and signature sizes of MiRitH in bytes.

7.2.2 Runtime

We provide an optimized implementation using AVX2 for MiRitH. The key generation, signature generation, and verification benchmarks were performed on an 11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz (Turbo Boost disabled). The reported timings are the medians and are shown in Table 5, where they are given in CPU cycles. This implementation uses the AVX2 optimized SHA3/SHAKE implementation from the *eXtended Keccak Code Package* (XKCP) [4].

[4]https://github.com/XKCP/XKCP

Set	Variant	Key generation	Signing	Verification
Ia	fast	108,903	8,703,311	7,311,069
	short		76,549,995	76,874,731
Ib	fast	199,556	8,015,345	7,558,761
	short		65,630,977	65,551,641
IIIa	fast	246,740	22,485,807	18,431,919
	short		192,858,411	175,520,472
IIIb	fast	373,836	24,538,474	22,470,437
	short		242,531,804	204,853,275
Va	fast	508,607	36,361,915	36,665,342
	short		308,565,196	310,604,452
Vb	fast	693,941	38,659,453	38,122,610
	short		327,068,513	330,632,038

Table 5: Median time (in clock cycles) for the AVX2 implementation of MiRitH on an 11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz (Turbo Boost disabled).

Additionally, we provide an optimized implementation for the ARM architecture using the NEON instruction set. All benchmarks were performed on an Apple M1 Max CPU, which are listed in Table 6. This implementation also leverages the XKCP for its SHA3/SHAKE implementation.

Set	Variant	Key generation	Signing	Verification
Ia	fast	84,260	4,802,648	4,463,499
	short		42,901,724	42,711,128
Ib	fast	159,018	6,380,382	5,933,225
	short		51,492,738	51,801,192
IIIa	fast	207,518	11,234,568	10,390,944
	short		94,497,277	94,227,698
IIIb	fast	315,849	13,284,941	12,349,311
	short		112,205,153	112,013,158
Va	fast	432,562	23,891,660	22,245,016
	short		196,659,024	194,568,129
Vb	fast	599,738	28,311,245	26,328,290
	short		241,600,195	240,977,263

Table 6: Median time (in clock cycles) for the NEON implementation of MiRitH on an Apple M1 Max.

7.2.3 Memory

We performed an analysis of the maximum memory usage of MiRitH by employing the massif tool of valgrind. The results are reported in Table 7. As valgrind is only available on Linux, we could not perform the memory analysis on our Apple machine, but we expect no significant difference in stack usage.

Set	Variant	Key generation	Signing	Verification
Ia	fast short	11,392	132,632 1,018,552	31,768 101,144
Ib	fast short	20,096	161,816 1,193,304	44,024 127,416
IIIa	fast short	23,184	286,376 2,196,552	64,616 157,032
IIIb	fast short	34,120	318,600 2,386,248	78,952 180,712
Va	fast short	46,488	516,984 3,815,864	116,424 230,680
Vb	fast short	64,216	570,296 4,116,984	140,408 265,272

Table 7: Memory usage (in bytes) for the AVX2 implementation of MiRitH. Benchmarks were done a 12th Gen Intel(R) Core(TM) i5-1240P.

7.3 MiRitH-Hypercube

7.3.1 Signature sizes

From Section 3.4, it follows that the maximum size (in bits) for a MiRitH-Hypercube signature is equal to

$$\underbrace{6\lambda}_{\mathsf{salt},h_1,h_2} + \tau \left(\underbrace{(k + r(n-r) + s(n-r) + sr) \cdot \log q}_{[\![\boldsymbol{\alpha}^{(\ell)}]\!]_N, [\![K^{(\ell)}]\!]_N, [\![C^{(\ell)}]\!]_N, [\![S^{(\ell)}]\!]_{i^*,(\ell)}} + \underbrace{\lambda \cdot D \cdot \log_2 N}_{(\mathsf{seed}_i^{(\ell)})_{i \neq i^*}} + \underbrace{2\lambda}_{\mathsf{com}_{i^*,(\ell)}} \right),$$

while the average size (in bits) is equal to

$$\underbrace{6\lambda}_{\mathsf{salt},h_1,h_2} + \tau \left(\underbrace{\frac{N-1}{N}}_{\Pr[i^{*,(\ell)} \neq N]} \cdot \underbrace{\big(k + r(n-r) + s(n-r)\big) \log q}_{[\![\boldsymbol{\alpha}^{(\ell)}]\!]_N, [\![K^{(\ell)}]\!]_N, [\![C^{(\ell)}]\!]_N} + \underbrace{sr \log q}_{[\![S^{(\ell)}]\!]_{i^*}} + \underbrace{\lambda \cdot D \cdot \log N}_{(\mathsf{seed}_i^{(\ell)})_{i \neq i^{*},(\ell)}} + \underbrace{2\lambda}_{\mathsf{com}_{i^*,(\ell)}} \right),$$

where s is the MPC protocol Π_s parameter, N the number of parties in one MPC protocol, D the hypercube dimension and τ the number of parallel repetitions, respectively. The values N and τ are set to achieve a soundness error smaller than $2^{-\lambda}$.

Table 8 shows the sizes of the public key, secret key, maximum signature and average signature for all the proposed parameter sets.

Set	Variant	Public key	Secret key	Signature (max)	Signature (avg)
Ia	fast	129	16	7,877	6,151
	short			5,673	4,661
	shorter			5,036	4,256
	shortest			4,536	3,936
Ib	fast	144	16	9,105	6,668
	short			6,309	4,894
	shorter			5,491	4,406
	shortest			4,886	4,051
IIIa	fast	205	24	17,139	13,372
	short			12,440	10,294
	shorter			10,746	9,131
	shortest			9,954	8,679
IIIb	fast	205	24	18,459	13,839
	short			13,136	10,512
	shorter			11,202	9,236
	shortest			10,314	8,762
Va	fast	253	32	31,468	23,888
	short			21,795	17,738
	shorter			19,393	16,254
	shortest			17,522	15,107
Vb	fast	274	32	34,059	25,006
	short			23,182	18,337
	shorter			20,394	16,663
	shortest			18,292	15,422

Table 8: Public keys, secret key, and signature sizes of MiRitH-Hypercube in bytes.

7.3.2 Runtime

As for MiRitH, we provide an optimized implementation using AVX2 for MiRitH-Hypercube. The key generation, signature generation, and verification benchmarks were performed on an 11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz (Turbo Boost disabled). The reported timings are the medians and are shown in Table 9, where they are given in CPU cycles.

Set	Variant	Key generation	Signing			Verification
			Offline	Online	Total	
Ia	fast	108,967	5,014,715	2,231,369 (30.79%)	7,246,084	6,061,955
	short		39,008,720	2,211,987 (5.37%)	41,220,707	40,976,634
	shorter		453,105,917	2,387,676 (0.52%)	455,493,593	456,564,597
	shortest		6,105,561,868	2,555,425 (0.04%)	6,108,117,293	6,195,562,217
Ib	fast	202,734	6,965,263	2,497,293 (26.39%)	9,462,556	7,914,458
	short		39,624,870	2,461,270 (5.85%)	42,086,140	42,047,669
	shorter		456,056,236	2,615,400 (0.57%)	458,671,636	450,442,537
	shortest		6,104,832,926	2,755,892 (0.05%)	6,107,588,818	6,040,512,181
IIIa	fast	247,821	10,651,817	4,920,028 (31.60%)	15,571,845	13,030,031
	short		65,262,560	4,989,171 (7.10%)	70,251,731	67,659,810
	shorter		717,628,079	5,477,766 (0.76%)	723,105,845	708,260,445
	shortest		9,855,053,596	5,987,719 (0.06%)	9,861,041,315	9,651,788,738
IIIb	fast	369,388	13,121,055	5,263,559 (28.63%)	18,384,614	15,550,479
	short		66,628,946	5,184,457 (7.22%)	71,813,403	75,999,541
	shorter		721,516,599	5,728,444 (0.79%)	727,245,043	732,036,291
	shortest		9,872,024,719	6,218,012 (0.06%)	9,878,242,731	10,063,176,060
Va	fast	521,018	23,230,321	10,014,703 (30.12%)	33,245,024	28,269,718
	short		109,614,396	8,879,212 (7.49%)	118,493,608	113,191,871
	shorter		1,280,920,702	10,065,728 (0.78%)	1,290,986,430	1,272,158,929
	shortest		16,697,326,803	10,718,196 (0.06%)	16,708,044,999	16,535,614,637
Vb	fast	709,957	29,363,137	11,302,559 (27.79%)	40,665,696	34,718,714
	short		128,369,712	10,127,974 (7.31%)	138,497,686	138,624,970
	shorter		1,348,619,656	10,848,403 (0.80%)	1,359,468,059	1,278,699,748
	shortest		17,453,659,114	11,355,820 (0.07%)	17,465,014,934	16,113,973,548

Table 9: Median time (in clock cycles) for the AVX2 implementation of MiRitH-Hypercube on an 11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz (Turbo Boost disabled).

We also provide an optimized implementation using the NEON instruction set for MiRitH-Hypercube. The key generation, signature generation, and verification benchmarks were performed on an Apple M1 Max chip. The reported timings are the medians and are shown in Table 10, where they are given in CPU cycles.

Set	Variant	Key generation	Signing			Verification
			Offline	Online	Total	
Ia	fast	83,870	2,857,835	1,276,938 (30.88%)	4,134,773	3,353,098
	short		22,725,008	1,241,771 (5.18%)	23,966,779	23,119,796
	shorter		251,449,323	1,346,319 (0.53%)	252,795,642	249,284,476
	shortest		3,120,765,642	1,381,872 (0.04%)	3,122,147,514	3,066,303,314
Ib	fast	158,960	3,850,299	1,407,544 (26.77%)	5,257,843	4,383,691
	short		23,875,525	1,370,267 (5.43%)	25,245,792	24,238,069
	shorter		255,023,842	1,493,766 (0.58%)	256,517,608	254,888,937
	shortest		3,183,130,055	1,502,461 (0.05%)	3,184,632,516	3,156,445,671
IIIa	fast	206,532	6,307,315	2,671,595 (29.75%)	8,978,910	8,218,136
	short		38,294,963	2,752,720 (6.71%)	41,047,683	40,022,357
	shorter		403,325,122	3,198,090 (0.79%)	406,523,212	403,676,867
	shortest		5,145,953,903	3,366,899 (0.07%)	5,149,320,802	5,120,266,642
IIIb	fast	315,414	7,292,093	2,890,753 (28.39%)	10,182,846	9,078,766
	short		39,895,520	2,896,027 (6.77%)	42,791,547	42,146,649
	shorter		414,978,504	3,227,023 (0.77%)	418,205,527	415,146,141
	shortest		5,275,047,923	3,381,685 (0.06%)	5,278,429,608	5,250,561,031
Va	fast	408,548	12,080,451	5,330,531 (30.62%)	17,410,982	14,759,906
	short		64,592,058	4,785,141 (6.90%)	69,377,199	67,830,786
	shorter		786,861,350	5,765,282 (0.73%)	792,626,632	800,576,501
	shortest		9,724,295,156	5,988,836 (0.06%)	9,730,283,992	9,799,923,065
Vb	fast	602,409	15,187,771	6,049,044 (28.48%)	21,236,815	18,167,108
	short		77,288,217	5,397,998 (6.53%)	82,686,215	80,615,500
	shorter		792,358,024	5,959,268 (0.75%)	798,317,292	800,096,979
	shortest		9,760,548,061	6,134,286 (0.06%)	9,766,682,347	9,811,165,545

Table 10: Median time (in clock cycles) for the NEON-ARM implementation of MiRitH-Hypercube on an Apple M1 Max chip.

7.3.3 Memory

An analysis of the maximum memory usage of MiRitH-Hypercube, performed using the massif tool of valgrind, is reported in Table 11. As for the MiRitH version, we did not perform the memory analysis on our Apple machine.

Set	Variant	Key generation	Signing	Verification
Ia	fast	11,344	86,040	29,752
	short		225,496	37,688
	shorter		1,777,816	221,624
	shortest		21,045,208	3,169,760
Ib	fast	20,144	107,288	41,080
	short		245,880	47,992
	shorter		1,798,424	231,768
	shortest		21,065,688	3,179,776
IIIa	fast	23,136	185,560	61,112
	short		501,400	70,072
	shorter		3,892,408	344,664
	shortest		47,345,912	4,767,136
IIIb	fast	34,104	209,560	74,096
	short		524,696	82,360
	shorter		3,915,064	356,920
	shortest		47,368,984	4,779,232
Va	fast	46,376	341,560	113,528
	short		877,048	118,456
	shorter		7,095,256	483,776
	shortest		84,164,344	6,379,800
Vb	fast	64,272	380,504	136,344
	short		915,160	138,712
	shorter		7,134,328	504,408
	shortest		84,203,416	6,399,072

Table 11: Memory usage (in bytes) for the AVX2 implementation of MiRitH-Hypercube. Benchmarks were done on a 12th Gen Intel(R) Core(TM) i5-1240P.

8 Advantages and limitations

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

Security In contrast to trapdoor-constructions, MiRitH is based on random instances of the MinRank problem. Hence, there is no additional structure introduced that could be exploited by future attacks. Further, due to its many applications in cryptanalysis, the MinRank problem is well-established in cryptographic applications and its hardness is well-understood.

Simplicity While the general framework of MPCitH introduces some overhead, our scheme breaks down into simple components. The used MPC protocol is based on a linear secret-sharing scheme, making input transformations and operations on shares straightforward. Moreover, at the lower level, our scheme relies only on standard linear algebra computations, making it easy to understand and implement.

Efficiency As MiRitH relies on the lower level only on linear-algebra computations on rather small matrices over finite fields, it is easy to optimize. However, the general ZKPoK and MPCitH context involves a lot of hashing operations, especially for computing commitments, which might become the bottleneck at some point.

Signature + Public Key Size As a ZKPoK construction, MiRitH offers very competitive signature sizes. Further, it uses very small public keys, especially after applying the outlined compression technique. The signature + public key metric, therefore, is mainly a signature size metric for MiRitH. However, again, inherent to the construction is a certain amount of commitments to be exchanged. Therefore, even by compressing problem related objects as far as possible, signatures can not become arbitrarily small.

Extension to ring signature A *ring signature scheme* enables a user to sign a message so that a *ring* of possible signers (of which the user is a member) is identified, without revealing exactly which member of that ring actually generated the signature [BKM09]. MiRitH, as any MinRank-based scheme, can be easily extended to a ring signature scheme (see, e.g., [BESV22, Section 5] for the details).

9 Security Proofs

9.1 Definitions

For the sake of completeness, we list below some common cryptographic definitions which will be used in the subsequent security proofs.

Definition 1 (Collision-Resistant Hash Function)**.** A function $H : \{0,1\}^* \to \{0,1\}^{p(\lambda)}$, with $p(\lambda)$ polynomially-bounded by λ, is a *collision-resistant hash function* if it can be computed in polynomial time and for every probabilistic polynomial algorithm $\mathcal{A}$ there exists a negligible function ε such that

$$\Pr\big[(x_1, x_2) \leftarrow \mathcal{A}(1^\lambda, H) : x_1 \neq x_2, H(x_1) = H(x_2)\big] < \varepsilon(\lambda).$$

Definition 2 ((t,ε)-Indistinguishability)**.** Two probabilistic distributions $\{D\}$ and $\{E\}$ are said to be (t,ε)*-indistinguishable*, where D, E, t, ε are functions of λ, if for every probabilistic algorithm $\mathcal{A}$ running in time at most t we have that

$$\big|\Pr\big[1 \leftarrow \mathcal{A}(x) : x \leftarrow D\big] - \Pr\big[1 \leftarrow \mathcal{A}(x) : x \leftarrow E\big]\big| < \varepsilon.$$

Definition 3 (Pseudorandom Generator (PRG))**.** Let $G : \{0,1\}^* \to \{0,1\}^*$ be a function such that for any $s \in \{0,1\}^\lambda$ we have $G(s) \in \{0,1\}^{p(\lambda)}$, where $p(\lambda)$ is a polynomially-bounded. We say that G is a (t,ε)*-secure pseudorandom generator* if $p(\lambda) > \lambda$ and the distributions

$$\big\{G(s) : s \xleftarrow{\$} \{0,1\}^\lambda\big\} \quad \text{and} \quad \big\{r \mid r \xleftarrow{\$} \{0,1\}^{p(\lambda)}\big\}$$

are (t,ε)-indistinguishable.

Seed Tree Pseudorandom Generator (TreePRG) A seed tree PRG is a function that constructs a binary tree on the input of the number of leaves and a root seed value in $\{0,1\}^\lambda$. Recursively, the seed value of each internal node is expanded using a PRG : $\{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ to generate the seed values of its children nodes. The main advantage of having a seed tree of N leaves is to reduce the communication cost when all but one of the leaf seeds needs to be revealed. In this case, only the seeds of a set of $\log N$ nodes are revealed, instead of $N-1$ as if the seeds were generated separately. For a formal definition of seed trees and their security analysis, we refer to [BKP20, Section 2.6].

We make use of a *commitment scheme* $\mathsf{Com} : \{0,1\}^* \times \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ that is assumed to be *computational hiding* and *computational binding*. We give these formal definitions below.

Definition 4 ((t,ε)-Hiding)**.** A commitment scheme Com is (t,ε)*-hiding* if for every pair of messages (m_1, m_2) the following distributions are (t,ε)-indistinguishable

$$\left\{c : c \leftarrow \mathsf{Com}(m_1,\rho),\ \rho \xleftarrow{\$} \{0,1\}^\lambda\right\} \quad \text{and} \quad \left\{c : c \leftarrow \mathsf{Com}(m_2,\rho),\ \rho \xleftarrow{\$} \{0,1\}^\lambda\right\}.$$

Definition 5 (Computational binding)**.** A commitment scheme Com is *computational binding* if for every algorithm $\mathcal{A}$ running in a time polynomial in λ we have that

$$\Pr\left[\mathsf{Com}(m_1,\rho_1) = \mathsf{Com}(m_2,\rho_2) : (m_1,\rho_1,m_2,\rho_2) \leftarrow \mathcal{A}(1^\lambda)\right] < \varepsilon(\lambda),$$

where ε is a negligible function.

9.2 MPC protocol

Proposition 2. *If $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$ then Π_s always outputs **accept**. If $\boldsymbol{M}_{\boldsymbol{\alpha}}^L \neq \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$ then Π_s outputs **accept** with probability at most $1/q^s$.*

Proof. The proof is similar to that of [Fen22, Lemma 1], but we include it for completeness. First, we have that

$$S = \sum_{i=1}^N [\![S]\!]_i = \sum_{i=1}^N \left(R \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^R]\!]_i + [\![A]\!]_i\right) = R \cdot \boldsymbol{M}_{\boldsymbol{\alpha}}^R + A.$$

Consequently, also using the fact that $C = A \cdot K$, we get that

$$\begin{aligned} V = \sum_{i=1}^N [\![V]\!]_i &= \sum_{i=1}^N \left(S \cdot [\![K]\!]_i - R \cdot [\![\boldsymbol{M}_{\boldsymbol{\alpha}}^L]\!]_i - [\![C]\!]_i\right) \\ &= S \cdot K - R \cdot \boldsymbol{M}_{\boldsymbol{\alpha}}^L - C = (R \cdot \boldsymbol{M}_{\boldsymbol{\alpha}}^R + A) \cdot K - R \cdot \boldsymbol{M}_{\boldsymbol{\alpha}}^L - A \cdot K \\ &= R \cdot (\boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K - \boldsymbol{M}_{\boldsymbol{\alpha}}^L). \end{aligned}$$

Hence, if $\boldsymbol{M}_{\boldsymbol{\alpha}}^L = \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$ then Π_s always outputs **accept**.

In contrast, suppose that $\boldsymbol{M}_{\boldsymbol{\alpha}}^L \neq \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$. Let $\Delta := \boldsymbol{M}_{\boldsymbol{\alpha}}^L - \boldsymbol{M}_{\boldsymbol{\alpha}}^R \cdot K$. Then, by the Rouché–Capelli theorem, the number of matrices $R \in \mathbb{F}_q^{s\times m}$ such that $R \cdot \Delta = \mathbf{0}$ is equal to $q^{(m-\mathsf{Rank}(\Delta^\intercal))s}$. Since $\Delta \neq \mathbf{0}$, we have that $\mathsf{Rank}(\Delta^\intercal) \geq 1$. Consequently, the probability that Π_s outputs **accept** is at most $1/q^s$. $\square$

9.3 ZKPoK for MinRank

Theorem 1 (Completeness)**.** *The ZKPoK of Fig. 3 is complete. That is, an honest prover will always convince an honest verifier.*

Proof. The completeness follows easily from the first part of Proposition 2. □

Theorem 2 (Soundness)**.** *Assume that the commitment scheme* Com *is binding and that the hash function* Hash_1 *is collision-resistant. Then the ZKPoK of Fig. 3 is sound with soundness error*

$$\varepsilon := \frac{1}{q^s} + \left(1 - \frac{1}{q^s}\right) \cdot \frac{1}{N}. \tag{7}$$

Precisely, suppose that there exists an efficient prover $\tilde{\mathcal{P}}$ *that, on input* $\boldsymbol{M}$*, convinces an honest verifier* V *to accept with probability* $\tilde{\varepsilon} > \varepsilon$*. Then there exists an efficient probabilistic extractor algorithm* $\mathcal{E}$ *that, given rewindable black-box access to* $\tilde{\mathcal{P}}$*, produces* $(\boldsymbol{\alpha}, K)$ *such that* $\boldsymbol{M}^L_{\boldsymbol{\alpha}} = \boldsymbol{M}^R_{\boldsymbol{\alpha}} \cdot K$ *by making an average number of calls to* $\tilde{\mathcal{P}}$ *that is upper bounded by*

$$\frac{4}{\tilde{\varepsilon} - \varepsilon}\left(1 + \frac{\tilde{\varepsilon} \ln 4}{\tilde{\varepsilon} - \varepsilon}\right).$$

Proof. We follow the soundness proof by Feneuil, Joux, and Rivain [FJR22]. For simplicity, we assume that the commitment scheme is perfectly binding since otherwise, if it is only computationally binding, we would have to deal with commitment collisions.

Let $\tilde{\mathcal{P}}$, $\tilde{\varepsilon}$, and ε be as in the statement of Theorem 2. In Fig. 19, we describe an extractor $\mathcal{E}$ that, employing the efficient prover $\tilde{\mathcal{P}}$, finds a witness $(\boldsymbol{\alpha}, K)$ satisfying $\boldsymbol{M}^L_{\boldsymbol{\alpha}} = \boldsymbol{M}^R_{\boldsymbol{\alpha}} \cdot K$.

```
$\mathcal{E}(\tilde{\mathcal{P}}, \tilde{\varepsilon})$
1 : for $n = 1, 2, \ldots$ do
2 :   Run $\tilde{\mathcal{P}}$ with the honest verifier $\mathsf{V}$ until it obtains a valid transcript $T$,
3 :   and save the randomness $x$ that generated $h_1$.
4 :   for $k = 1, 2, \ldots, \lceil (\ln 4)/(\tilde{\varepsilon} - \varepsilon) \rceil$ do
5 :     Run $\tilde{\mathcal{P}}(x)$ with $\mathsf{V}$ until it obtains a valid transcript $T'$ with $i^* \neq i^{*\prime}$,
6 :     and save the corresponding $(\boldsymbol{\alpha}, K)$.
7 :     if $\boldsymbol{M}^L_{\boldsymbol{\alpha}} = \boldsymbol{M}^R_{\boldsymbol{\alpha}} \cdot K$ then
8 :       return $(\boldsymbol{\alpha}, K)$
```

Figure 19: Extractor $\mathcal{E}$.

Our goal is to estimate the average number of calls that $\mathcal{E}$ makes to $\tilde{\mathcal{P}}$ before returning. Let $\eta := (\tilde{\varepsilon} - \varepsilon)/(2\tilde{\varepsilon})$ and note that $(1 - \eta)\tilde{\varepsilon} > \varepsilon$. Denote by $\mathsf{Succ}_{\tilde{\mathcal{P}}}$ the event that $\tilde{\mathcal{P}}$ succeeds on convincing an honest verifier V. Also, let X be the random variable that samples the randomness used by $\tilde{\mathcal{P}}$ in the generation of the initial commitment h_1. We say that an x in the sample space of X is *good* if we have

$$\Pr\left[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x\right] \geq (1 - \eta)\tilde{\varepsilon}.$$

By the Splitting Lemma [PS00], we have that

$$\Pr\big[x \text{ is good} \mid X = x \text{ and } \mathsf{Succ}_{\tilde{\mathcal{P}}}\big] \geq \eta. \tag{8}$$

Assume that $\mathcal{E}$ samples a successful transcript T with $X = x$, and assume that x is good. By definition, we have that

$$\Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x\big] \geq (1-\eta)\tilde{\varepsilon} > \varepsilon > \frac{1}{N},$$

This implies there exists a successful transcript T' with $X = x$ and $i^* \neq i^{*\prime}$.

Note that, in the verifications of the two transcripts T and T', the revealed shares of $[\![\boldsymbol{\alpha}]\!]$ must be equal, as well as the revealed shares of $[\![K]\!]$. Otherwise, since the commitment scheme is assumed to be perfectly binding, we would have $h_1 = h_1'$, contrary to the assumption that Hash_1 is collision-resistant. Therefore, there exists a unique and well-defined witness $(\boldsymbol{\alpha}, K)$ corresponding to the transcripts T and T'.

Let $(\boldsymbol{\alpha}, K)$ be the witness generated by $\tilde{\mathcal{P}}$ when $X = x$. Let us prove that if x is good then $\boldsymbol{M}^L_{\boldsymbol{\alpha}} = \boldsymbol{M}^R_{\boldsymbol{\alpha}} \cdot K$. We proceed by proving the contraposition, that is, we assume that $\boldsymbol{M}^L_{\boldsymbol{\alpha}} \neq \boldsymbol{M}^R_{\boldsymbol{\alpha}} \cdot K$ and we prove that x is not good. Let FP denote the event that a genuine execution of the MPC protocol outputs a false positive, i.e., a zero matrix V. Then, from Proposition 2, we have that $\Pr[\mathsf{FP}] \leq 1/q^s$. Hence, we get that

$$\begin{aligned}\Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x\big] &= \Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \text{ and } \mathsf{FP} \mid X = x\big] + \Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \text{ and } \overline{\mathsf{FP}} \mid X = x\big] \\ &\leq \frac{1}{q^s} + \left(1 - \frac{1}{q^s}\right) \cdot \Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x \text{ and } \overline{\mathsf{FP}}\,\big].\end{aligned} \tag{9}$$

Note that the event $\mathsf{Succ}_{\tilde{\mathcal{P}}}$ requires that the sharing $[\![V]\!]$ in the first response of the prover must encode a zero matrix, while the event $\overline{\mathsf{FP}}$ implies that a genuine execution outputs a non-zero matrix V. Hence, to have a successful transcript, the prover must cheat for the simulation of at least one party. If the prover cheats for several parties, there is no way it can produce a successful transcript. If the prover cheats for exactly one party (among the N parties), the probability of being successful is at most $1/N$. Therefore, we have that

$$\Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x \text{ and } \overline{\mathsf{FP}}\,\big] \leq \frac{1}{N},$$

which together with (9) yields

$$\Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x\big] \leq \frac{1}{q^s} + \left(1 - \frac{1}{q^s}\right) \cdot \frac{1}{N} = \varepsilon,$$

that is, x is not good, as desired.

If x is good, then the probability that the k-th iteration of the inner loop of Fig. 19 finds a transcript T' is

$$\begin{aligned}&\Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \text{ and } i^* \neq i^{*\prime} \mid X = x\big] = \Pr\big[\mathsf{Succ}_{\tilde{\mathcal{P}}} \mid X = x\big] - \Pr\big[i^* = i^{*\prime}\big] \\ &\geq (1-\eta)\tilde{\varepsilon} - \frac{1}{N} \geq (1-\eta)\tilde{\varepsilon} - \varepsilon = \frac{\tilde{\varepsilon} - \varepsilon}{2}.\end{aligned}$$

Therefore, the inner loop finds T' with a probability of at least

$$1 - \left(1 - \frac{\tilde{\varepsilon} - \varepsilon}{2}\right)^{\lceil (\ln 4)/(\tilde{\varepsilon} - \varepsilon)\rceil} \geq 1 - \exp\left((\ln 2) \ln\left(1 - \frac{\tilde{\varepsilon} - \varepsilon}{2}\right) / \frac{\tilde{\varepsilon} - \varepsilon}{2}\right) > 1 - e^{-\ln 2} = \frac{1}{2},$$

where we used the inequality $\ln(1-x)/x < -1$, which holds for $x \in (0,1)$.

Now we can give an upper bound for the average number of calls that $\mathcal{E}$ makes to $\tilde{\mathcal{P}}$ before terminating. First, $\mathcal{E}$ makes an average number of $1/\tilde{\varepsilon}$ calls to obtain a valid transcript T. Then, in light of (8), the probability that x is good is at least η. Moreover, by the previous consideration, if x is good then $\mathcal{E}$ finds T' with probability at least $1/2$ after $\lceil(\ln 4)/(\tilde{\varepsilon}-\varepsilon)\rceil$ calls to $\tilde{P}$. Therefore, the probability that T' is found is at least $\eta/2$.

Putting all together, we get that the average number of calls of the extractor $\mathcal{E}$ to $\tilde{\mathcal{P}}$ is upper-bounded by

$$\left(\frac{1}{\tilde{\varepsilon}} + \left\lceil \frac{\ln 4}{\tilde{\varepsilon}-\varepsilon} \right\rceil\right) \cdot \frac{2}{\eta} < \frac{4}{\tilde{\varepsilon}-\varepsilon}\left(1 + \frac{\tilde{\varepsilon}\ln 4}{\tilde{\varepsilon}-\varepsilon}\right).$$

The proof is complete. □

Theorem 3 (Honest-Verifier Zero-Knowledge)**.** *In the ZKPoK of Fig. 3, if the PRG is $(t, \varepsilon_{\mathrm{PRG}})$-secure and the commitment scheme is $(t, \varepsilon_{\mathsf{com}})$-hiding, then there exists an efficient simulator that outputs transcripts $(t, \varepsilon_{\mathrm{PRG}} + \varepsilon_{\mathsf{com}})$-indistinguishable from real transcripts of the protocol.*

Proof. As in the proof of soundness, we follow the approach by Feneuil, Joux, and Rivain [FJR22]. In Fig. 20, we describe a simulator Simulator that we will prove that outputs transcripts that are $(t, \varepsilon_{\mathrm{PRG}} + \varepsilon_{\mathsf{com}})$-indistinguishable from real transcripts of the protocol.

We begin by considering the subroutine InternalSimulator and by proving that its responses are $(t, \varepsilon_{\mathrm{PRG}})$-indistinguishable from the responses of an honest prover for the same challenges (R, i^*). In order to do so, we describe the following sequence of simulators.

Simulator 1 (Actual protocol)**.** This simulator, described in Fig. 21, outputs h_2, $(\mathsf{state}_i, \rho_i)_{i \neq i^*}$, and $[\![S]\!]_{i^*}$ from the transcript of a genuine execution of the protocol with a prover that knowns a witness $(\boldsymbol{\alpha}, K)$ and receives a challenge (R, i^*).

Simulator 2. This simulator is identical to Simulator 1, except that for party i^* it uses true randomness instead of seed-derived randomness. Precisely, $[\![A]\!]_{i^*}$ is generated by true randomness, and if $i^* \neq N$ then $[\![\boldsymbol{\alpha}]\!]_{i^*}$, $[\![K]\!]_{i^*}$, $[\![C]\!]_{i^*}$ are generated by true randomness. It is easy to see that the probability of distinguishing Simulator 2 and Simulator 1 in running time t is no more than $\varepsilon_{\mathrm{PRG}}$.

Simulator 3. This simulator is identical to Simulator 2, but $[\![\boldsymbol{\alpha}]\!]_N$, $[\![K]\!]_N$, $[\![C]\!]_N$ are generated by true randomness and $[\![V]\!]_{i^*} = -\sum_{i \neq i^*} [\![V]\!]_i$. In particular, the output does not depend anymore on the witness $(\boldsymbol{\alpha}, K)$, and Simulator 3 takes as input only the challenges (R, i^*).

Now we have to prove that the distributions of the outputs of Simulator 2 and Simulator 3 are in fact identical.

If $i^* = N$, then the changes only impact the shares $[\![S]\!]_N$, $[\![V]\!]_N$. Since the additive term $[\![A]\!]_N$ is sampled uniformly at random in Simulator 2, and it does not change in Simulator 3, we have that the distribution of $[\![S]\!]_N$ does not change. Moreover, since $[\![V]\!]_N = -\sum_{i \neq N} [\![V]\!]_i$ already in Simulator 2, its distribution does not change in Simulator 3.

If $i^* \neq N$, then the changes impact first $\mathsf{aux} = ([\![\boldsymbol{\alpha}]\!]_N, [\![K]\!]_N, [\![C]\!]_N)$, then $[\![S]\!]_N$ and $[\![V]\!]_N$, since they are derived from aux, and finally $[\![V]\!]_{i^*}$, since it has the additive term $[\![V]\!]_N$. However, aux was already uniformly random in Simulator 2. Indeed, the shares in aux are computed by adding share values from parties $i \neq N$, including the i^* party (which is uniformly random in Simulator 2). Therefore, the output distributions of Simulator 2 and Simulator 3 are identical.

```
InternalSimulator$(R, i^*)$
1 :  $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda$,   $(\mathsf{seed}_i, \rho_i)_{i\in[N]} \leftarrow \mathrm{TreePRG}(\mathsf{seed})$
2 :  For each party $i \in [N] \setminus \{i^*\}$
3 :    $[\![A]\!]_i \leftarrow \mathrm{PRG}(\mathsf{seed}_i)$
4 :    if $i \neq N$ then
5 :      $[\![\boldsymbol{\alpha}]\!]_i, [\![K]\!]_i, [\![C]\!]_i \leftarrow \mathrm{PRG}(\mathsf{seed}_i)$,   $\mathsf{state}_i \leftarrow \mathsf{seed}_i$
6 :    else
7 :      $([\![\boldsymbol{\alpha}]\!]_N, [\![K]\!]_N, [\![C]\!]_N) \xleftarrow{\$} \mathbb{F}_q^k \times \mathbb{F}_q^{r\times(n-r)} \times \mathbb{F}_q^{s\times(n-r)}$
8 :      $\mathsf{aux} \leftarrow ([\![\boldsymbol{\alpha}]\!]_N, [\![K]\!]_N, [\![C]\!]_N)$,   $\mathsf{state}_N \leftarrow (\mathsf{seed}_N, \mathsf{aux})$
9 :    $[\![S]\!]_i \leftarrow R \cdot [\![\boldsymbol{M}^R_{\boldsymbol{\alpha}}]\!]_i + [\![A]\!]_i$
10 :   $[\![V]\!]_i \leftarrow S \cdot [\![K]\!]_i - R \cdot [\![\boldsymbol{M}^L_{\boldsymbol{\alpha}}]\!]_i - [\![C]\!]_i$
11 : $[\![S]\!]_{i^*} \xleftarrow{\$} \mathbb{F}_q^{s\times r}$
12 : $[\![V]\!]_{i^*} \leftarrow -\sum_{i\neq i^*} [\![V]\!]_i$
13 : $h_2 \leftarrow \mathrm{Hash}_2([\![S]\!]_1, [\![V]\!]_1, \ldots, [\![S]\!]_N, [\![V]\!]_N)$
     return $\big(h_2, (\mathsf{state}_i, \rho_i)_{i\neq i^*}, [\![S]\!]_{i^*}\big)$
```

```
Simulator()
1 :  $R \xleftarrow{\$} \mathbb{F}_q^{s\times m}$,   $i^* \xleftarrow{\$} [N]$
2 :  $\big(h_2, (\mathsf{state}_i, \rho_i)_{i\neq i^*}, [\![S]\!]_{i^*}\big) \leftarrow \mathsf{InternalSimulator}(R, i^*)$
3 :  $\mathsf{com}_i \leftarrow \mathsf{Com}(\mathsf{state}_i, \rho_i)$ for each $i \in [N] \setminus \{i^*\}$
4 :  $\mathsf{com}_{i^*} \xleftarrow{\$} \{0,1\}^{2\lambda}$
5 :  $h_1 \leftarrow \mathrm{Hash}_1(\mathsf{com}_1, \ldots, \mathsf{com}_N)$
6 :  $\mathsf{rsp} \leftarrow \big((\mathsf{state}_i, \rho_i)_{i\neq i^*}, \mathsf{com}_{i^*}, [\![S]\!]_{i^*}\big)$
     return $(h_1, R, h_2, i^*, \mathsf{rsp})$
```

Figure 20: HVZK simulator.

Simulator 4 (Internal HVZK simulator)**.** This is InternalSimulator in Fig. 20. The only difference with Simulator 3 is that $[\![S]\!]_{i^*}$ is drawn uniformly at random. As explained before, this does not impact the output distribution.

Thus we have shown that InternalSimulator outputs responses that are $(t, \varepsilon_{\mathrm{PRG}})$-indistinguishable from the responses of the real protocol on the same challenges of an honest verifier. Applying the hiding property of the commitment scheme on com_{i^*}, we obtain that Simulator in Fig. 20 outputs a transcript that is $(t, \varepsilon_{\mathrm{PRG}} + \varepsilon_{\mathsf{com}})$-indistinguishable from a real transcript of the protocol. □

9.4 Unforgeability

Theorem 4. *Suppose that* PRG *is $(t, \varepsilon_{\mathrm{PRG}})$-secure and that any adversary running in time t has at most an advantage $\varepsilon_{\mathrm{MR}}$ against the underlying MinRank problem associated with a public key $\boldsymbol{M}$. Moreover, assume that* Hash_0, Hash_1, *and* Hash_2 *are modeled as random oracles. Let $\mathcal{A}$ be an adaptive chosen-message adversary against the signature scheme described in Fig. 5, running*

```
InternalSimulator$_1(\boldsymbol{\alpha}, K, R, i^*)$
1:  $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\lambda$,   $(\mathsf{seed}_i, \rho_i)_{i\in[N]} \leftarrow \mathrm{TreePRG}(\mathsf{seed})$
2:  For each party $i \in [N]$
3:    $[\![A]\!]_i \leftarrow \mathrm{PRG}(\mathsf{seed}_i)$
4:    if $i \neq N$ then
5:      $[\![\boldsymbol{\alpha}]\!]_i, [\![K]\!]_i, [\![C]\!]_i \leftarrow \mathrm{PRG}(\mathsf{seed}_i)$,   $\mathsf{state}_i \leftarrow \mathsf{seed}_i$
6:    else
7:      $[\![\boldsymbol{\alpha}]\!]_N \leftarrow \boldsymbol{\alpha} - \sum_{i\neq N}[\![\boldsymbol{\alpha}]\!]_i$,   $[\![K]\!]_N \leftarrow \boldsymbol{\alpha} - \sum_{i\neq N}[\![K]\!]_i$,   $[\![C]\!]_N \leftarrow A \cdot K - \sum_{i\neq N}[\![C]\!]_i$
8:      $\mathsf{aux} \leftarrow ([\![\boldsymbol{\alpha}]\!]_N, [\![K]\!]_N, [\![C]\!]_N)$,   $\mathsf{state}_N \leftarrow (\mathsf{seed}_N, \mathsf{aux})$
9:    $[\![S]\!]_i \leftarrow R \cdot [\![\boldsymbol{M}^R_{\boldsymbol{\alpha}}]\!]_i + [\![A]\!]_i$
10:   $[\![V]\!]_i \leftarrow S \cdot [\![K]\!]_i - R \cdot [\![\boldsymbol{M}^L_{\boldsymbol{\alpha}}]\!]_i - [\![C]\!]_i$
11: $h_2 \leftarrow \mathrm{Hash}_2([\![S]\!]_1, [\![V]\!]_1, \ldots, [\![S]\!]_N, [\![V]\!]_N)$
    return $\big(h_2, (\mathsf{state}_i, \rho_i)_{i\neq i^*}, [\![S]\!]_{i^*}\big)$
```

Figure 21: $\mathsf{InternalSimulator}_1$ on input the witness $(\boldsymbol{\alpha}, K)$ and a challenge (R, i^*).

in time t, making q_s signing queries, and q_i queries to Hash_i *for $i = 0, 1, 2$. Then $\mathcal{A}$ succeeds in outputting a valid forgery with probability*

$$\Pr[\mathsf{Forge}] \leq \frac{3(q_0+q_1+q_2+\tau N q_s)^2}{2^{2\lambda+1}} + \frac{q_s(q_s+q_0+q_1+q_2)}{2^{2\lambda}} + q_s\tau\varepsilon_{\mathrm{PRG}} + \varepsilon_{\mathrm{MR}} + \max_{0\leq t\leq\tau} P(t), \tag{10}$$

where

$$P(t) := \Big(1 - \big(1 - P_b(t)\big)^{q_1}\Big)\Big(1 - \big(1 - P_{a,c}(t)\big)^{q_2}\Big),$$
$$P_b(t) := \left(\frac{1}{q^s}\right)^t \left(1 - \frac{1}{q^s}\right)^{\tau - t} \binom{\tau}{t},$$
$$P_{a,c}(t) := \frac{1}{N^{\tau-t}}.$$

Proof. In the following, we will define a sequence of experiments involving $\mathcal{A}$. We let $\Pr_i[\cdot]$ denote the probability of an event in the ith experiment. Let t denote the running time of the entire experiment (including $\mathcal{A}$'s running time, the time required to answer signing queries, and the time to verify $\mathcal{A}$'s output).

Experiment 1. This is the interaction of $\mathcal{A}$ with the real signature scheme. Precisely, first KeyGen is run to obtain $\boldsymbol{M}, \boldsymbol{\alpha}, K$. Then $\mathcal{A}$ is given the public key $\boldsymbol{M}$ and access to the signing oracle, which returns a valid signature under the key $(\boldsymbol{\alpha}, K)$ for any requested message, and to the random oracles Hash_i for $i = 0, 1, 2$. At the end of this experiment, $\mathcal{A}$ outputs a message-signature pair (msg, σ). We let Forge denote the event that msg was not previously queried by $\mathcal{A}$ to the signing oracle and that σ is a valid signature for msg.

Our goal is to upper-bound $\Pr_1[\mathsf{Forge}]$.

Experiment 2. We proceed as Experiment 1 with the only exception that we abort if, during the course of the experiment, a collision of Hash_0, Hash_1, or Hash_2 is found. The number of queries to Hash_0, Hash_1, or Hash_2, by either the adversary or the signing oracle, is at most $q_0+q_1+q_2+\tau N q_s$. Therefore, using the classic bound for the probability of a collision of a hash function[5], we have that

$$\left|\mathrm{Pr}_1[\mathsf{Forge}] - \mathrm{Pr}_2[\mathsf{Forge}]\right| < \frac{3(q_0+q_1+q_2+\tau N q_s)^2}{2^{2\lambda+1}}.$$

Experiment 3. We proceed as Experiment 2 with the only exception that we abort if, while answering a signature query, the salt sampled in the signature query is equal to a salt used in a previous query (by $\mathcal{A}$ or by another signing query) to Hash_0, Hash_1 or Hash_2. For each signature query, the probability to abort is at most $(q_s+q_0+q_1+q_2)/2^{2\lambda}$. Therefore, we have that

$$\left|\mathrm{Pr}_2[\mathsf{Forge}] - \mathrm{Pr}_3[\mathsf{Forge}]\right| \leq \frac{q_s(q_s+q_0+q_1+q_2)}{2^{2\lambda}}.$$

Experiment 4. The difference with Experiment 3 is that, when signing a message msg, in Phases 2 and 4 of the signing algorithm (see Fig. 5) h_1 and h_2 are set to be uniformly random values in $\{0,1\}^{2\lambda}$, without making the corresponding queries to Hash_1 and Hash_2.

Hence, the outcome of Experiment 3 differs from this one only if, in the course of answering a signing query, it happens that the underlying query to Hash_1 or Hash_2 was made before by $\mathcal{A}$ or by a signing query. But this cannot happen, since in such a case Experiment 3 would abort. Thus, we have that

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

Experiment 5. The difference with Experiment 4 is that, for each $\ell \in [\tau]$, we set $\mathsf{com}^{(\ell)}_{i^*,(\ell)}$ to be a uniformly random value in $\{0,1\}^{2\lambda}$, without making the corresponding query to Hash_0.

Hence, the outcome of Experiment 4 differs from this one only if, in the course of answering a signing query, Hash_0 receives an input that was previously queried. Note that such an event cannot occur within the same signing query, since the indices i and ℓ are part of the input to Hash_0, and it cannot occur from a previous query (signing query or $\mathcal{A}$ query), because in such a case Experiment 3 would abort. Therefore, we have that

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

Experiment 6. This experiment is obtained by modifying Experiment 5 as follows. For $\ell \in [\tau]$, the signer uses the code of $\mathsf{InternalSimulator}$ in Fig. 20 to generate the parties' views in one execution of Phases 1 and 3. We write $\mathsf{InternalSimulator}_{\mathsf{salt}}$ to denote a call to this simulator that prepends salt to the sampled seed in input to TreePRG and PRG. Thus, signature queries are now answered as described in Fig. 22. Observe that the secret $(\boldsymbol{\alpha}, K)$ is no longer used for generating signatures. Recall from the proof of Theorem 3 that an adversary against $\mathsf{InternalSimulator}_{\mathsf{salt}}$ has a distinguishing advantage $\varepsilon_{\mathrm{PRG}}$ (corresponding to execution time t) since commitments are built outside of the simulator. Consequently, we have that

$$\left|\mathrm{Pr}_5[\mathsf{Forge}] - \mathrm{Pr}_6[\mathsf{Forge}]\right| \leq q_s \tau \varepsilon_{\mathrm{PRG}}.$$

[5]The probability that n calls to a hash functions $H : \{0,1\}^* \to \{0,1\}^{2\lambda}$ lead to a collision is at most $n(n-1)/2^{2\lambda+1}$.

```
$\mathcal{O}^{\mathsf{Sign}'}(\mathsf{msg})$
     Phase 0.
1 :  $\mathsf{salt} \xleftarrow{\$} \{0,1\}^{2\lambda}, \quad h_1 \xleftarrow{\$} \{0,1\}^{2\lambda}, \quad h_2 \xleftarrow{\$} \{0,1\}^{2\lambda}$
2 :  $R^{(1)}, \dots, R^{(\tau)} \leftarrow \mathrm{PRG}(h_1)$ where $R^{(\ell)} \in \mathbb{F}_q^{s \times m}$.
3 :  $i^{*,(1)}, \dots, i^{*,(\tau)} \leftarrow \mathrm{PRG}(h_2)$ where $i^{*,(\ell)} \in [N]$.
     Phases 1 and 3.
4 :  For each $\ell \in [\tau]$:
5 :      $\Big( (\mathsf{state}_i^{(\ell)})_{i \neq i^{*,(\ell)}}, \Big( [\![S^{(\ell)}]\!]_i, [\![V^{(\ell)}]\!]_i \Big)_{i \in [N]} \Big) \leftarrow \mathsf{InternalSimulator}_{\mathsf{salt}}(R^{(\ell)}, i^{*,(\ell)})$
6 :      $\mathsf{com}_i^{(\ell)} \leftarrow \mathrm{Hash}_0(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)})$ for each $i \neq i^*$.
7 :      $\mathsf{com}_{i^{*,(\ell)}}^{(\ell)} \xleftarrow{\$} \{0,1\}^{2\lambda}$
     Phases 2 and 4.
8 :  Set $\mathrm{Hash}_1\big(\mathsf{msg}, \mathsf{salt}, (\mathsf{com}_i^{(\ell)})_{i \in [N], \ell \in [\tau]}\big)$ equal to $h_1$ (i.e., if $\mathcal{A}$ makes
     subsequent queries to $\mathrm{Hash}_1$ with the same input, then return $h_1$ as output).
9 :  Set $\mathrm{Hash}_2\big(\mathsf{msg}, \mathsf{salt}, ([\![S^{(\ell)}]\!]_i, [\![V^{(\ell)}]\!]_i)_{i \in [N], \ell \in [\tau]}\big)$ equal to $h_2$ (i.e., if $\mathcal{A}$ makes
     subsequent queries to $\mathrm{Hash}_2$ with the same input, then return $h_2$ as output).
     Phase 5: Assembling the signature.
10 : $\sigma \leftarrow \Big( \mathsf{salt}, h_1, h_2, \Big( (\mathsf{state}_i^{(\ell)})_{i \neq i^{*,(\ell)}}, \mathsf{com}_{i^{*,(\ell)}}^{(\ell)}, [\![S^{(\ell)}]\!]_{i^{*,(\ell)}} \Big)_{\ell \in [\tau]} \Big)$
     return $\sigma$
```

Figure 22: Signing query for a message msg in Experiment 6.

Experiment 7. At any point during this experiment, we say that we have a $\mathsf{correct}$ execution ℓ^* if, in a query to Hash_2 with input

$$\Big(\mathsf{msg}, \mathsf{salt}, h_1, \Big(\left[\!\!\left[S^{(\ell)}\right]\!\!\right]_i, \left[\!\!\left[V^{(\ell)}\right]\!\!\right]_i \Big)_{i \in [N],\, \ell \in [\tau]}\Big),$$

we have that:

1. there is a previous query $h_1 \leftarrow \mathrm{Hash}_1\big(\mathsf{msg}, \mathsf{salt}, \mathsf{com}_1^{(1)}, \dots, \mathsf{com}_N^{(\tau)}\big)$;
2. each $\mathsf{com}_i^{(\ell^*)}$ is the output of a previous query to Hash_0 (by either $\mathcal{A}$ or the signing oracle) with input $\big(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell^*)}\big)$;
3. and a solution $(\boldsymbol{\alpha}, K)$ to the MinRank instance $\boldsymbol{M}$ can be extracted from $(\mathsf{state}_i^{(\ell^*)})_{i \in [N]}$.

In this experiment, in each query made by $\mathcal{A}$ to Hash_2 (where msg was not previously queried), it is checked if there is a correct execution. We call this event Solve. Note that if Solve occurs then $(\mathsf{state}_i^{(\ell^*)})_{i \in [N]}$ (which can be determined from the oracle queries of $\mathcal{A}$) allow one to efficiently recover a solution $(\boldsymbol{\alpha}, K)$ to the MinRank instance $\boldsymbol{M}$. Thus, we have that $\Pr_6[\mathsf{Solve}] \leq \varepsilon_{\mathrm{MR}}$.

Hence, we get that

$$\begin{aligned}\Pr_6[\mathsf{Forge}] &= \Pr_6[\mathsf{Forge} \text{ and } \mathsf{Solve}] + \Pr_6[\mathsf{Forge} \text{ and not } \mathsf{Solve}]\\ &\le \varepsilon_{\mathrm{MR}} + \Pr_6[\mathsf{Forge} \text{ and not } \mathsf{Solve}].\end{aligned}$$

Now, suppose that the adversary produced a forgery (msg, σ) and Solve did not occur. In particular, it does not occur for the query to Hash_2 of the form

$$\Big(\mathsf{msg}, \mathsf{salt}, h_1, \Big(\big[\!\big[S^{(\ell)}\big]\!\big]_i, \big[\!\big[V^{(\ell)}\big]\!\big]_i \Big)_{i\in[N],\ \ell\in[\tau]}\Big)$$

associated to the forgery (msg, σ). Since there is a forgery then the above condition 1 verifies, that is, there is a previous query $h_1 \leftarrow \mathrm{Hash}_1\big(\mathsf{msg}, \mathsf{salt}, \mathsf{com}_1^{(1)}, \dots, \mathsf{com}_N^{(\tau)}\big)$. Therefore, for each $\ell \in [\tau]$, we have that exactly one of the three following cases must occur:

(a) $\mathsf{com}_{i^*,\ell}^{(\ell)}$ is not the output of a query to Hash_0 with an input of the form $\big(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell^*)}\big)$.

(b)
- for all $i \in [N]$, $\mathsf{com}_i^{(\ell)}$ is the output of a query to Hash_0 with input $\big(\mathsf{salt}, \ell, i, \mathsf{state}_i^{(\ell)}\big)$,
- the witness $(\boldsymbol{\alpha}, K)$ extracted from $(\mathsf{state}_i^{(\ell)})_{i\in[N]}$ is a not a solution to $\boldsymbol{M}$,
- $\big[\!\big[S^{(\ell)}\big]\!\big]_{i^*}$ and $\big[\!\big[V^{(\ell)}\big]\!\big]_{i^*}$ are genuinely computed from $\mathsf{state}_{i^*}^{(\ell)}$, h_1, and $S^{(\ell)}$.

(c) Same as (b) but $\big[\!\big[S^{(\ell)}\big]\!\big]_{i^*}$ or $\big[\!\big[V^{(\ell)}\big]\!\big]_{i^*}$ are not genuinely computed from $\mathsf{state}_{i^*}^{(\ell)}$, h_1, and $S^{(\ell)}$.

Clearly, if (b) occurs for a round $\ell \in [\tau]$, this means that the MPC protocol Π_s in Fig. 2 to verify matrix-multiplication triple is honestly followed by each party $i \in [N]$. Hence, from Proposition 2, we have that the adversary has probability $1/q^s$ to have (b) satisfied for this round ℓ. Therefore, the probability of having exactly $t \in [\tau]$ rounds satisfying (b) is at most

$$P_b(t) = \left(\frac{1}{q^s}\right)^t \left(1 - \frac{1}{q^s}\right)^{\tau - t} \binom{\tau}{t}.$$

If (b) does not occur for a round $\ell \in [\tau]$, this means that any other second challenge obtained from h_2 different from $i^{*,\ell}$ would make the forgery fail. Hence, the probability of having this round ℓ not leading to rejection is at most $1/N$. Therefore, the probability of having exactly $\tau - t \in [\tau]$ rounds satisfying (a) or (c) is at most

$$P_{a,c}(t) = \frac{1}{N^{\tau - t}}.$$

In view of the above, the probability of having Forge and not Solve with exactly t rounds satisfying (b) after q_1 queries to Hash_1 and q_2 queries to Hash_2 is at most

$$P(t) = \Big(1 - \big(1 - P_b(t)\big)^{q_1}\Big)\Big(1 - \big(1 - P_{a,c}(t)\big)^{q_2}\Big).$$

Thus, we have

$$\Pr_6[\mathsf{Forge} \text{ and not } \mathsf{Solve}] \le \max_{0\le t\le \tau} P(t).$$

The final upper bound (10) follows by putting together all the bounds for the probabilities given in the experiments. □

References

[AMGH$^+$23] C. Aguilar-Melchor, N. Gama, J. Howe, A. Hülsing, D. Joseph, and D. Yue. The Return of The SDitH. In *Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V*, page 564–596, Berlin, Heidelberg, 2023. Springer-Verlag.

[ARZV22] G. Adj, L. Rivera-Zamarripa, and J. 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] M. Bardet and M. Bertin. Improvement of algebraic attacks for solving superdetermined MinRank instances. In *Post-Quantum Cryptography. PQCrypto 2022.*, volume 13512 of *Lecture Notes in Computer Sciences*, pages 107–123. Springer, Cham., 2022.

[BBB$^+$22] M. Bardet, P. Briaud, M. Bros, P. Gaborit, and J. P. 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] M. Bardet, M. Bros, D. Cabarcas, P. Gaborit, R. Perlner, D. Smith-Tone, J.-P. Tillich, and J. Verbel. Improvements of algebraic attacks for solving the rank decoding and MinRank problems. In *Advances in cryptology—ASIACRYPT 2020. Part I*, volume 12491 of *Lecture Notes in Computer Sciences*, pages 507–536. Springer, Cham., 2020.

[BCH$^+$23] W. Beullens, M.-S. Chen, S.-H. Hung, M. J. Kannwischer, B.-Y. Peng, C.-J. Shih, and B.-Y. Yang. Oil and vinegar: Modern parameters and implementations. Cryptology ePrint Archive, Paper 2023/059, 2023. https://eprint.iacr.org/2023/059.

[BESV22] E. Bellini, A. Esser, C. Sanna, and J. Verbel. MR-DSS - Smaller MinRank-Based (Ring-)Signatures. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022*, volume 13512 of *Lecture Notes in Computer Science*, pages 144–169. Springer, 2022.

[BKM09] A. Bender, J. Katz, and R. Morselli. Ring signatures: Stronger definitions, and constructions without random oracles. *J. Cryptology*, 22(1):114–138, 2009.

[BKP20] W. Beullens, S. Katsumata, and F. Pintore. Calamari and Falafl: logarithmic (linkable) ring signatures from isogenies and lattices. In *Advances in cryptology—ASIACRYPT 2020. Part II*, volume 12492 of *Lecture Notes in Comput. Sci.*, pages 464–492. Springer, Cham, 2020.

[Cou01] N. T. Courtois. Efficient zero-knowledge authentication based on a linear algebra problem MinRank. In *Advances in cryptology—ASIACRYPT 2001 (Gold Coast)*, volume 2248 of *Lecture Notes in Computer Sciences*, pages 402–421. Springer, Berlin, 2001.

[Div14] NIST Computer Security Division. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS Publication 202, National Institute of Standards and Technology, U.S. Department of Commerce, May 2014.

[DSS23] A. J. Di Scala and C. Sanna. Smaller public keys for MinRank-based schemes. arXiv preprint, 2023. https://arxiv.org/abs/2302.12447.

[FEDS13] J. C. Faugère, M. S. El Din, and P. J. Spaenlehauer. On the complexity of the generalized minrank problem. *J. Symb. Comput.*, 55:30–58, 2013.

[Fen22] T. 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.

[FJR22] T. Feneuil, A. Joux, and M. Rivain. Syndrome decoding in the head: Shorter signatures from zero-knowledge proofs. In *Advances in Cryptology – CRYPTO 2022. CRYPTO 2022.*, volume 13508 of *Lecture Notes in Computer Science*, pages 541–572. Springer, Cham., 2022.

[FS87] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO 1986*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[GC00] L. Goubin and N. T. Courtois. Cryptanalysis of the TTM Cryptosystem. In *Advances in Cryptology — ASIACRYPT 2000*, pages 44–57. Springer Berlin Heidelberg, 2000.

[GD22] H. Guo and J. Ding. Algebraic relation of three minrank algebraic modelings. In S. Mesnager and Z. Zhou, editors, *Arithmetic of Finite Fields - 9th International Workshop, WAIFI 2022, Chengdu, China, August 29 - September 2, 2022, Revised Selected Papers*, volume 13638 of *Lecture Notes in Computer Science*, pages 239–249. Springer, 2022.

[KS99] A. Kipnis and A. Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 19–30, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KZ20] D. Kales and G. Zaverucha. An attack on some signature schemes constructed from five-pass identification schemes. In S. Krenn, H. Shulman, and S. Vaudenay, editors, *Cryptology and Network Security*, pages 3–22, Cham, 2020. Springer International Publishing.

[KZ22] D. Kales and G. Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. Cryptology ePrint Archive, Paper 2022/588, 2022. https://eprint.iacr.org/2022/588.

[NIS] NIST. Post-Quantum Cryptography – Security (Evaluation Criteria). https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria). Accessed: March 15, 2023.

[PS00] D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[VBC$^+$19] J. Verbel, J. Baena, D. Cabarcas, R. Perlner, and D. Smith-Tone. On the complexity of “superdetermined” minrank instances. In J. Ding and R. Steinwandt, editors, *Post-Quantum Cryptography. PQCrypto 2019*, pages 167–186, 2019.

[Zal99] C. Zalka. Grover’s quantum searching algorithm is optimal. *Phys. Rev. A*, 60:2746–2751, Oct 1999.