Université Paris-Saclay
UFR Sciences, UVSQ
Master 2, 1er semestre 2024-2025

Rapport de Projet – Cryptographie et Programmation C

Schéma de signature post-quantique MiRitH

Auteurs :
COUTE Maxime
MAGOIS Jules

Encadrants :
M. ROTELLA Yann

Remis le 15 février 2025

SOMMAIRE

Première partie

I/- Théorie

1.1 Problème MinRank

Le problème MinRank est un problème algorithmique fondamental en cryptographie post-quantique et en théorie des codes. Il joue un rôle clé dans l'analyse de la sécurité de plusieurs systèmes cryptographiques, notamment ceux basés sur les codes linéaires et les systèmes multivariés.
On le définit ainsi :

Problème MinRank : Soit $\mathbb{K}$ un corps, $m, n \in \mathbb{N}$ et $r < n$, on considère les m matrices $n \times n$ sur $\mathbb{K}$ $M_1, \ldots, M_m$.

Trouver une combinaison linéaire $\alpha \in \mathbb{K}^m$ telle que :

$$rg\left(\Sigma_i \alpha_i M_i\right) \leq r$$

L'importance du problème MinRank en optimisation combinatoire et donc notamment en cryptographie tient au fait que c'est un problème NP-difficile, ce qui signifie qu'il n'existe pas d'algorithme connu capable de le résoudre en temps polynomial dans le cas général. Il existe bien des cryptanalyses du problème MinRank, par exemple une attaque algébrique utilisant des bases de Gröbner pour résoudre les systèmes polynomiaux associés à une instance [1], mais il résiste bien encore aujourd'hui aux attaques et figure ainsi parmi les bases de choix sur lesquelles construire des schémas cryptographiques post-quantiques, comme l'a montré la dernière compétition du NIST appelant à plusieurs schémas reposant sur ce problème. On notera d'ailleurs que MiRitH, que nous étudions dans ce rapport, a été sélectionné le 25 octobre 2024 avec 13 autres candidats pour être évalué au second tour de la compétition pour la standardisation de schéma de signature post-quantique.

MiRitH [5] est donc basé sur la difficulté du problème MinRank : il s'agit de prouver la connaissance d'une solution, c'est-à-dire d'une combinaison linéaire non triviale de petit rang de matrices définies sur un corps fini satisfaisant la condition citée précédemment pour une instance $(M_1, \ldots, M_m)$ du problème MinRank. Cependant, on ne souhaite pas donner cette solution. On utilise donc une preuve à divulgation nulle de connaissance basée sur le framework MPCitH que l'on transforme en schéma de signature non interactif avec l'heuristique de Fiat-Shamir. Revenons en détail sur ces différentes notions.

1.2 Multi-Party Computation

1.2.1 MPC

Le Multi-Party Computation (MPC), ou calcul multipartite (sécurisé, on précisera Secure MPC), est une technique cryptographique qui permet à plusieurs parties de calculer conjointement une fonction sur leurs entrées respectives sans révéler ces entrées aux autres parties. Il permet donc à un groupe de participants de collaborer pour effectuer un calcul tout en gardant leurs données privées. Cela signifie que chaque participant peut contribuer à un calcul global sans avoir à partager ses données sensibles avec les autres participants ou un tiers de confiance.

Le fonctionnement du MPC repose sur plusieurs principes cryptographiques :

1. Définition de la Fonction : Les participants définissent ensemble la fonction qu'ils souhaitent cal-

[1] FAUGERE Jean-Charles, *« Cryptanalysis of MinRank »*, https://link.springer.com/chapter/10.1007/978-3-540-85174-5_16

culer. Cette fonction peut être simple (comme une addition) ou complexe (comme un algorithme de machine learning) ;

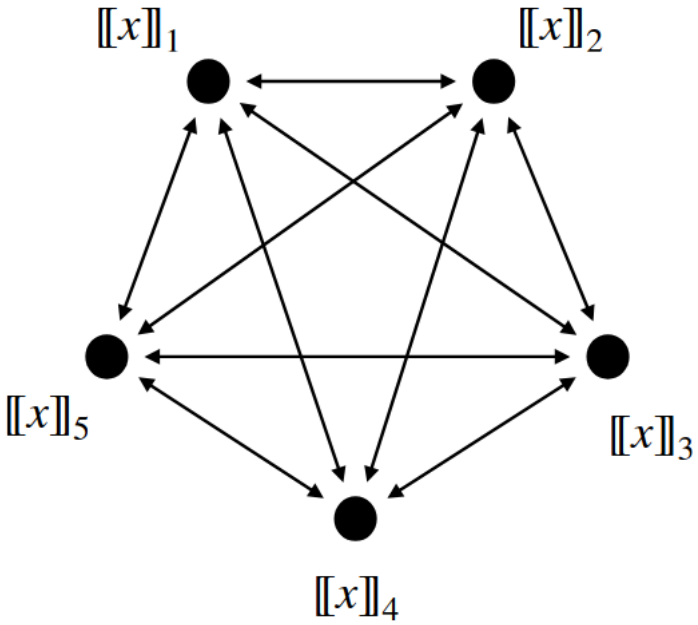
2. Distribution des Entrées : Chaque participant divise secrètement son entrée en plusieurs parts, appelées « shares ». Ces parts sont distribuées aux autres participants de manière à ce que chaque participant reçoive une part de chaque entrée ;

3. Calcul Sécurisé : Les participants effectuent des calculs intermédiaires sur les parts qu'ils possèdent. Ces calculs sont conçus de manière à ce que les résultats intermédiaires ne révèlent aucune information sur les entrées originales ;

4. Recombinaison des Résultats : Une fois les calculs intermédiaires terminés, les résultats sont combinés pour obtenir le résultat final de la fonction. Ce résultat peut être révélé à tous les participants ou seulement à certains d'entre eux, selon les besoins ;

5. Vérification : Des techniques de vérification peuvent être utilisées pour s'assurer que les calculs ont été effectués correctement et que personne n'a triché.

Reprenant les schémas de Thibauld Feneuil [3, 4], on représente ainsi un processus de calcul multipartite pour un exemple à 5 parties avec des parts additives telles que $x = \sum_{i=1}^{5} [[x_i]]$:



1.2.2 MPCitH

Le MPC-in-the-Head (MPCitH) est une approche innovante pour transformer des protocoles de calcul multipartite sécurisé (MPC) en preuves à divulgation nulle de connaissance (Zero-Knowledge Proofs of Knowledge, ZKPoP). Cette technique permet à une partie (le prouveur) de convaincre une autre partie (le vérifieur) qu'une déclaration est vraie sans révéler aucune information supplémentaire. Voici comment fonctionne le MPCitH :

L'idée principale derrière le MPCitH est de simuler un protocole MPC dans la tête du prouveur. Plutôt que d'avoir plusieurs parties réelles qui interagissent, le prouveur simule ces interactions lui-même.

Cela permet de créer une preuve qui peut être vérifiée par le vérifieur sans interaction directe avec les parties simulées.

1. Définition du Protocole MPC : Le prouveur commence par définir un protocole MPC pour la fonction qu'il veut prouver. Ce protocole est conçu pour être exécuté par plusieurs parties virtuelles ;

2. Simulation des Parties Virtuelles : Le prouveur simule plusieurs instances de lui-même, chacune agissant comme une partie distincte dans le protocole MPC. Ces parties virtuelles suivent le protocole MPC comme si elles étaient des participants réels ;

3. Génération des Preuves : Pendant la simulation, le prouveur génère des preuves pour chaque étape du protocole MPC. Ces preuves montrent que chaque partie virtuelle suit correctement le protocole ;

4. Commitment et Révélation : Le prouveur utilise des techniques de commitment pour s'engager sur les valeurs utilisées dans la simulation. Ensuite, il révèle certaines de ces valeurs de manière sélective pour convaincre le vérifieur que la simulation a été effectuée correctement ;

5. Vérification : Le vérifieur reçoit les commitments et les révélations du prouveur. Il vérifie que les révélations sont cohérentes avec les commitments et que le protocole MPC a été suivi correctement. Si tout est en ordre, le vérifieur est convaincu que la déclaration est vraie.

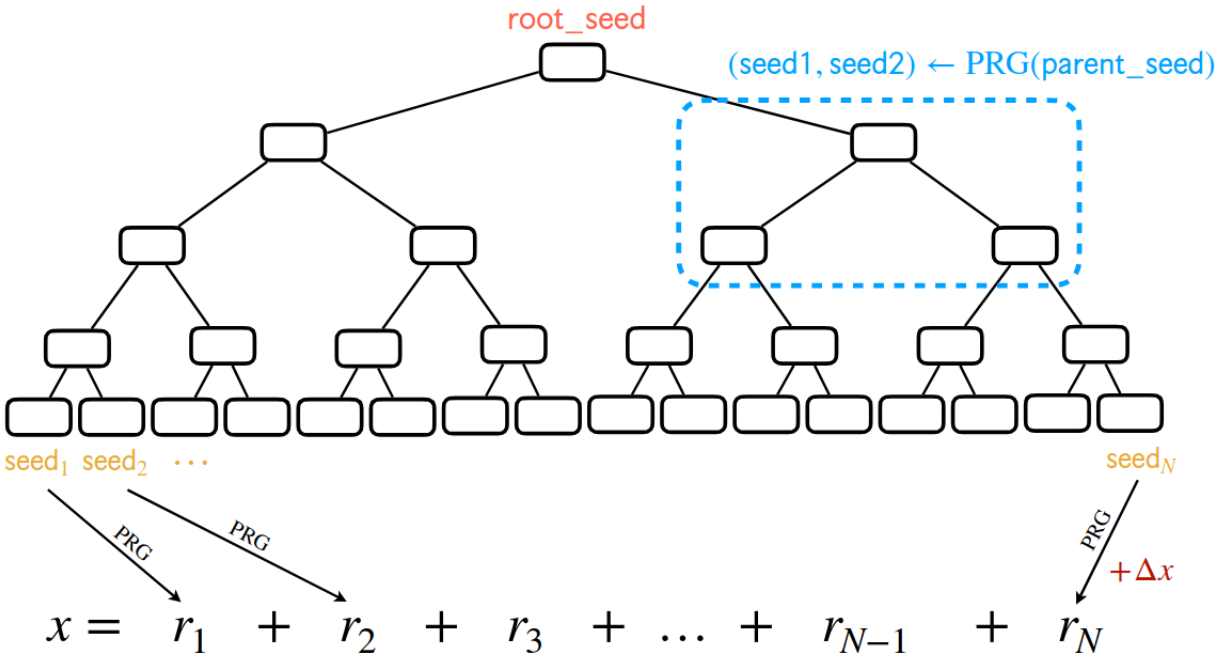
1.2.3 Seed Tree

Le concept de « Seed Tree » est utilisé dans MiRitH pour améliorer l'efficacité et la sécurité du processus de signature. Voici comment cela fonctionne :

— Structure Arborescente : Une structure arborescente est utilisée pour organiser les « seeds » (graines) utilisées dans la génération de la signature. Chaque nœud de l'arbre représente une graine ;

— Génération de Graines : Les graines sont générées de manière hiérarchique, chaque nœud parent générant des graines pour ses nœuds enfants. Cela permet de réduire la quantité de données aléatoires nécessaires pour générer une signature ;

— Sécurité et Efficacité : La structure arborescente permet de réutiliser certaines graines, ce qui améliore l'efficacité du processus de signature. De plus, elle garantit que les graines sont générées de manière sécurisée et imprévisible.



1.3 Preuve à divulgation nulle de connaissance

1.3.1 Définition

Une preuve à divulgation nulle de connaissance (Zero-Knowledge Proof of Knowledge, ZKPoK) est un protocole cryptographique qui permet à une partie (le prouveur) de convaincre une autre partie (le vérifieur) qu'une déclaration est vraie, sans révéler aucune information autre que la validité de cette déclaration. En d'autres termes, le prouveur prouve qu'il connaît une information sans révéler l'information elle-même.

Caractéristiques des Preuves à Divulgation Nulle de Connaissance [2] :

— Complétude (Completeness) : Si l'énoncé est vrai, un prouveur honnête peut convaincre un vérifieur honnête ;

— Solidité (Soundness) : Si l'énoncé est faux, aucun prouveur malhonnête ne peut convaincre un vérifieur honnête qu'il est vrai, sauf avec une probabilité négligeable ;

— Connaissance Nulle (Zero-Knowledge) : Si l'énoncé est vrai, le vérifieur n'apprend rien d'autre que le fait que l'énoncé est vrai.

Exemple :

Imaginons qu'Alice veuille prouver à Bob qu'elle connaît un code x sans révéler le code lui-même. Voici comment un protocole ZKPoK pourrait fonctionner :

1. Préparation : Alice connaît le code mais ne veut pas le révéler à Bob qui ne connaît pas x mais $g^x mod(p)$, avec p un nombre premier et g un générateur.

2. Engagement : Alice choisit un nombre aléatoire v et calcule un engagement $t = g^v mod(p)$.
Alice envoie t à Bob.

3. Bob envoie un défi aléatoire c à Alice.

4. Réponse : Alice envoie $r = v - cx$ à Bob.

5. Vérification : Bob calcule $g^r y^c$.

Analyse :

— Complétude : Si Alice connaît le code, elle peut toujours répondre correctement au défi de Bob.

— Solidité : Si Alice ne connaît pas le code, elle ne peut pas prédire le défi de Bob et donc ne peut pas tricher.

— Connaissance Nulle : Bob n'apprend rien sur le code lui-même.

1.3.2 Heuristique de Fiat-Shamir

L'exemple de preuve à divulgation nulle de connaissance précédent conduit à l'heuristique de Fiat-Shamir qui permet de transformer une preuve à divulgation nulle de connaissance en preuve non-interactive à divulgation nulle de connaissance :

Protocole Σ initial : On note $\mathcal{E}, \mathcal{C}$ et $\mathcal{R}$ respectivement l'espace des engagements, des challenges et des réponses. Pour prouver la connaissance de $(x, w) \in \mathcal{X} \times \mathcal{W}$, on a ainsi :

— Un engagement, durant lequel le prouveur envoie au vérifieur un élément $e \in \mathcal{E}$.

— Le challenge, ou défi, durant lequel le vérifieur répond un élément $c \in \mathcal{C}$.

— La réponse, durant laquelle le prouveur répond un élément $r \in \mathcal{R}$.

A partir de ce protocole, le prouveur simule la présence du vérifieur par une fonction de hachage modélisée comme un oracle aléatoire : $hash : \mathcal{E} \times \mathcal{X} \longrightarrow \mathcal{C}$.

— Le prouveur génère un élément $e \in \mathcal{E}$ comme dans le protocole interactif. Puis il hache $c = hash(e, x)$ et calcule une réponse r adéquate pour le challenge c avant d'envoyer $\pi = (e, r)$ comme preuve.

— Pour vérifier cette preuve, le vérifieur commence par hacher (e, x) pour obtenir c et vérifier que r est bien une réponse correcte pour le couple (e, c).

1.4 Application dans MiRitH

Soit $q, m, n, k, r \in \mathbb{N}^*$, avec q une puissance de nombre premier.
Etant donné le tuple $M = (M_0, \ldots, M_k) \in \left(\mathbb{F}_q^{m \times n}\right)^{k+1}$, pour résoudre le problème MinRank il nous faut trouver $\alpha = (\alpha_1, \ldots, \alpha_k) \in \mathbb{F}_q^k$ tel que :

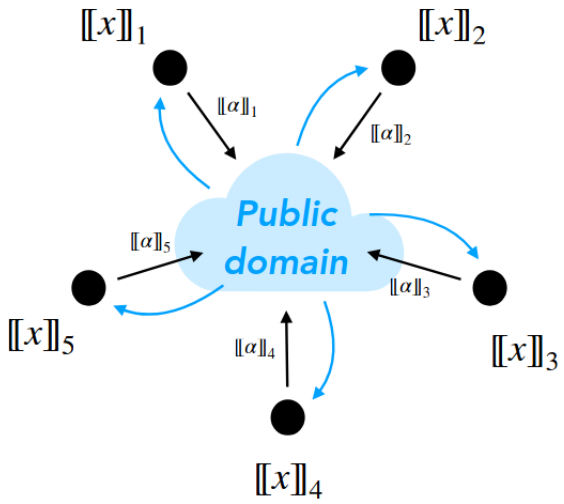
$$rg\left(M_0 + \sum_{i=1}^{k} \alpha_i M_i\right) \leq r.$$

D'après la modélisation Kipnis-Shamir, on a une solution lorsque l'égalité $M_\alpha^L = M_\alpha^R \cdot K$ est vérifiée, avec $M_\alpha = M_0 + \sum_{i=1}^{k} \alpha_i M_i$ et $M_\alpha = \left[M_\alpha^L | M_\alpha^R\right]$, $M_\alpha^L \in \mathbb{F}_q^{m \times (n-r)}$.

On utilise donc un protocole MPC pour prouver la connaissance d'une solution au problème MinRank pour une instance M donnée sans divulguer cette solution, donc le prouveur simule un protocole MPC

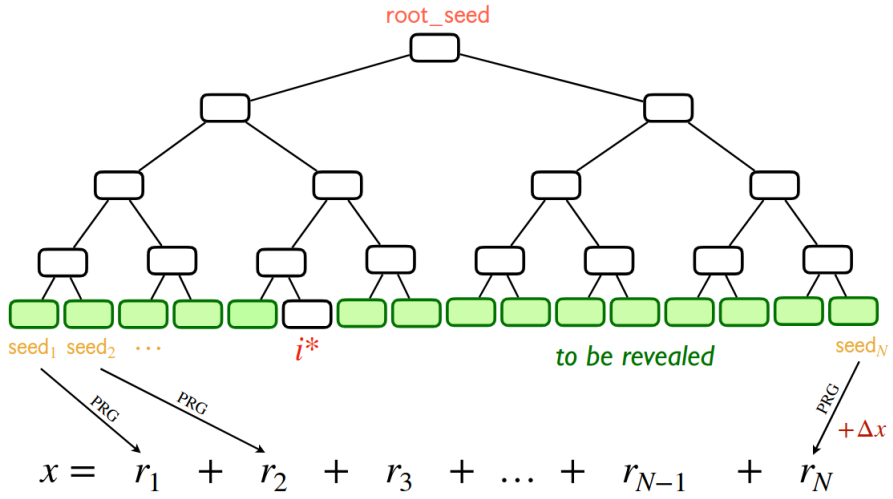
« dans sa tête » dans lequel on calcule en plusieurs parties $M_\alpha^L = M_\alpha^R \cdot K$ en partageant en parts additives α et K. Pour répartir en N parties, on tire donc aléatoirement $N-1$ valeurs $\alpha_1, \ldots, \alpha_{N-1}$ puis on calcule la dernière de sorte à bien avoir un partage additif : $\alpha_N = \alpha - \sum_{i=1}^{N-1} \alpha_i$.

— Le prouveur, après avoir généré les parts $[[x]] = \sum_{i=1}^{N} [[x]]_i$, envoie les commitments $Com^{\rho_i}([[x]]_i)$ et les évaluations des parts dans le protocole MPC $[[\alpha]]_i$ au vérifieur.

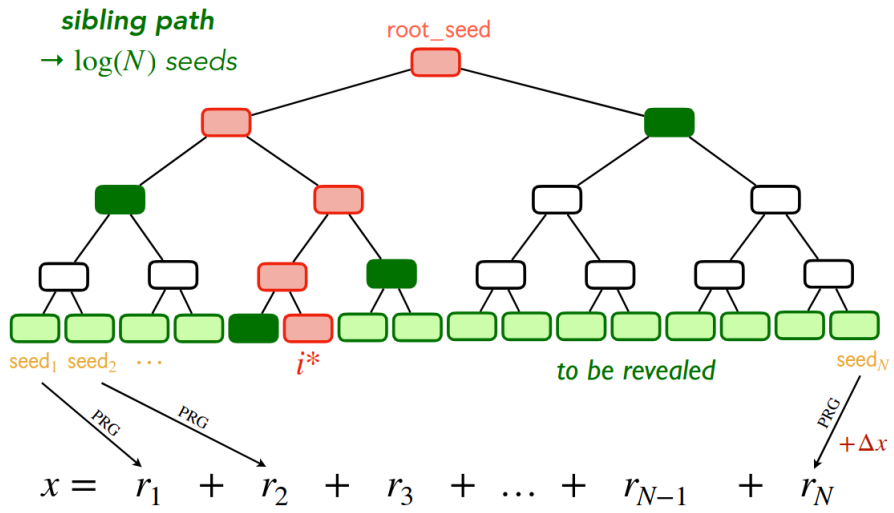


— Le vérifieur choisit une partie $i^\star, 1 \leq i^\star \leq N$ et l'envoie au prouveur.

— Le prouveur ouvre les parties $i \neq i^\star$ et envoie $([[x]]_i, \rho_i)_{i \neq i^\star}$ au vérifieur. C'est là que l'on se sert du seed tree que l'on a généré à partir d'une seed (`root_seed` sur le schéma suivant) et qui nous a permis de générer les seeds de toutes les parties :



Plus précisément, il faut révéler également les états intermédiaires de $i^\star$:



— Le vérifieur vérifie les commitments $Com^{\rho_i}([[x]]_i)$ et les calculs obtenus par le MPC $[[\alpha]]_i = \phi([[x]]_i), \quad \forall i \neq i^\star$ et, par linéarité de la multiplication matricielle, il peut ainsi vérifier $M_\alpha^L = M_\alpha^R \cdot K$, c'est-à-dire comme on l'a vu avec la modélisation Kipnis-Shamir que α est bien une solution du problème MinRank pour l'instance M sans avoir révélé α.

Un vérifieur souhaitant tricher pour retrouver la solution α peut retrouver les calculs de toutes les parties à partir des états initiaux pour tout $i \neq i^\star$ et vérifier avec les commitments. Le fait de conserver un état initial, celui de la partie $i^\star$, conduit à une probabilité de deviner la valeur de 1 sur le nombre de parties, c'est-à-dire $\frac{1}{N}$. En répétant un nombre τ de tours, on obtient finalement une probabilité d'erreur de $N^{-\tau}$, négligeable suivant le nombre de parties simulées et de tours effectués.

Deuxième partie

II/- Implémentation

2.1 Hiérarchie des fichiers

`types.h` contient tous les types utilisés dans le projet.

`constants.c` contient les table d'addition et multiplication dans $GF(16)$ et quelques paramètres de signature standards.

`field_arithmetics.c` définit l'addition et la multiplication dans $GF(16)$.

`matrix.c` définit les opérations sur les matrices.

`mpc.c` définit les fonctions pour effectuer le protocole MPC.

`packing.c` contient les fonctions pour compresser et décompresser plusieurs structures.

`random.c` contient toutes les fonctions en lien avec le PRNG.

`key_generation.c` contient principalement la fonction key_gen.

`seed_tree.c` contient les fonctions pour générer un arbre de graines.

`sign.c` contient (une partie) des fonctions pour signer un message.

test est un dossier contenant les différents tests.

2.2 Arithmétique dans $GF(16)$

Le fichier `field_arithmetics.c` contient les fonctions arithmétiques standards sur le corps $\mathbb{F}_{2^4}$:

1. `scalar_add` pour l'addition,
2. `scalar_mul` pour la multiplication.

Ces fonction ne font qu'appliquer une table `GF_16_ADD_TABLE` ou `GF_16_MUL_TABLE`, qui sont stockées dans le fichier `constants.c`. Ces tables ont été générées via les fonctions

1. `print_gf_16_addition_table`, et
2. `print_gf_16_mul_table`,

du fichier `field_arithmetics.c`.

2.3 Génération de clé

2.3.1 Structures de données

PrivateKey contient :

- `lambda` un entier positif,
- `seed` un tableau de booléens de taille `lambda`.

PublicKey contient :

- `lambda` un entier positif,
- `seed` un tableau de booléens de taille `lambda`,
- `m0` une matrice.

SignatureParameters contient :

- `lambda` un entier (la sécurité requise),
- `matrix_dimensions` une paire d'entiers (la taille de chaque matrice de l'instance MinRank),
- `field` un champ définissant le corps fini utilisé pour les calculs,
- `target_rank` un entier qui représente le rang r du problème MinRank,
- `solution_size` un entier qui représente le nombre de matrices à combiner dans l'instance MinRank,

`first_challenge_size` un entier qui représente le nombre de lignes de la matrice dans le premier défi.

PublicPrivateKeyPair contient :

`public_key` la clé publique,

`private_key` la clé privée.

2.3.2 La fonction `key_gen`

La fonction `key_gen` génère une paire clé publique / clé privée (une structure `PublicPrivateKeyPair`), en prenant comme argument un ensemble de paramètres `params` sous la forme d'une structure `SignatureParameters`. Elle procède en plusieurs étapes :

1. génération de la clé privée `private_key` via la fonction `generate_seed`,
2. génération de la graine publique via la même fonction,
3. création des deux générateurs pseudo-aléatoires `public_random_state` et `private_random_state`, initialisation avec leur graines respectives,
4. création d'un vecteur `alpha` et d'une matrice `sum` initialisée à 0,
5. pour chaque i de 1 à `params.solution_size` :
   (a) génération d'une matrice `m_i` avec `public_random_state`,
   (b) génération d'un scalaire `alpha[i]` avec `private_random_state`,
   (c) mise à jour `sum` $\longleftarrow$ `sum` + `alpha[i]` $\times$ `m_i`
6. génération de deux matrice aléatoire `K`, `E_R`
7. calcul du produit `E_L` = `K` $\times$ `E_R`, puis de leur concaténation `E` = `E_L` $||$ `E_R`,
8. calcul de la matrice `m0` = `E` $-$ `sum`,
9. la paire de clés finale `result` a :
   (a) pour clé publique : la graine de l'étape 2, la matrice `m0`,
   (b) pour clé privée : la graine de l'étape 1.

2.4 MPCitH

2.4.1 Structures de données

MinRankInstance contient :

`matrix_count` le nombre de matrices de l'instance,

`matrix_array` un tableau de `Matrix` de taille `matrix_count`.

MinRankSolution contient :

`matrix_count` le nombre de matrices de l'instance,

`target_rank` le rang de la combinaison linéaire de matrices,

`alpha` la solution au problème MinRank,

`K` le témoin de la solution α : une matrice $K \in \mathcal{M}_{r,n-r}(\mathbb{F}_{16})$ telle que pour l'instance $M_0, \ldots, M_k$ du problème MinRank, on ait :

$$\sum_i \alpha_i M_i \times \begin{pmatrix} I_{n-r} \\ K \end{pmatrix} = 0.$$

PartyState contient les données associées à chaque parti :

`M_left` un partage de la matrice $\sum \alpha_i M_i$ (partie gauche)

`M_left` un partage de la matrice $\sum \alpha_i M_i$ (partie droite)

`S` une matrice intermédiaire du protocol MPC,

`V` la matrice finale, à sommer sur tous les partis, ont dont la somme doit être zéro.

2.4.2 la fonction `mpc_check_solution`

Elle vérifie la `solution` d'une certaine `instance`, en simulant `number_of_parties` partis. Elle utilise un PRNG `random_state` pour générer des partages. Voici ce qu'elle effectue :

1. génération d'un tableau `parties` de `PartyState`,
2. partage de α et calcul de chaque `party.M_left, party.M_right`,
3. génération d'une matrice aléatoire A,
4. partage de A et mise à jour de chaque `party.S`,
5. sommation de tous les `party.S` pour obtenir la matrice S,
6. calcul de la matrice $C = AK$, où K provient de la `solution`,
7. partage de C et mise à jour de chaque `party.V`,
8. sommation de chaque `party.V` pour obtenir V
9. renvoi du booléen `V == 0`.

2.5 L'arbre de graines `tree_prg`

Ceci est implémenté dans le fichier `seed_tree.c`. Il se compose de cinq fonctions.

La fonction `PRG_init` a pour but de fournir une graine à un PRNG. Elle fournit la concaténation de ses deux arguments `salt, seed` au PRNG.

La fonction `PRG_bytes` utilise un PRNG pour remplir une variable d'octets aléatoires. La variable ainsi remplie est un tableau de `unsigned char`.

La fonction `TreePRG` crée un tableau de graines `tree`, vu comme un arbre parcouru dans l'ordre NLR (Node-Left-Right). Chaque noeud est une graine, utilisée pour générer ses deux graines filles. Enfin, la liste des graines que l'on recherche est celle des feuilles de l'arbre, autrement dit la seconde moitié du tableau `tree`.

La fonction `Tree_pack` est utilisée pour, à partir d'un arbre de graines `tree` et un index `i_star`, obtenir un ensemble de graines de taille $\log(N)$, où N est le nombre de feuilles de l'arbre. Cet ensemble à la propriété qu'il permet de retouver toutes les feuilles, sauf celle d'indice `i_star`.

La fonction `Tree_unpack` est justement la fonction qui permet de retrouver (presque) toutes les feuilles, à partir du résultat de la fonction précédente.

2.6 Signature

Phase 1 de la signature
La fonction `phase_one` réalise la première phase d'un processus de signature. Elle prend en entrée plusieurs variables vides qu'elle va remplir :

`commits` un tableau de dimension deux contenant les engagements de chaque parti pour chaque tour,

`data` un tableau de dimension deux contenant les données initiales associées à chaque parti,

ainsi que plusieurs variables déjà remplies :

`instance` une instance du problème MinRank,

`solution` une solution à l'instance précédente,

`params` l'ensemble des paramètres de la signature.

Voici les étapes détaillées de cette fonction :

1. Initialisation et génération de graines aléatoire : une graine est générée pour chaque tour, et cette graine est utilisée pour générer les graines de chaque parti

2. Pour les parties sauf la dernière (i! = `N_PARTIES` $- 1$) : Des matrices aléatoires $\underline{\alpha}_i, A_i, K_i, C_i$ sont générées. Un engagement est calculé et stocké dans `com[i]`.

3. Pour la dernière partie ($i ==$ `N_PARTIES` $- 1$) : Les parts additives $\underline{\alpha}_{N-1}, K_{N-1}$ sont calculées en soustrayant les parts des autres parties de la clé secrète α et K. La matrice `A` est ouverte en sommant toutes les parts `A_shr`. La part additive `C_shr`[`N_PARTIES` $- 1$] est calculée en multipliant `A` par `K` et en soustrayant les parts des autres parties. Un engagement avec des données auxiliaires est calculé et stocké dans `com[i]`.

Phase 2 de la signature
la fonction `phase_two` est prévue pour la deuxième phase. Elle commence par calculer un hash de `salt || seed || commits`, puis utilise ce hash pour créer les premiers défis (un défi par tour).

Phase 3 de la signature
La fonction `phase_three` est prévue pour réaliser la troisième phase du processus de signature. Elle prend en entrée des matrices publiques et secrètes, ainsi que les parts additives générées précédemment. L'implémentation de cette fonction repose principalement sur le protocole MPC, décrit précédemment et dont le code est développé dans le fichier mpc.c.

Phase 4 de la signature
La fonction `sign_phase4` réalise la quatrième phase du processus de signature. Elle prend en entrée `i_star` et un hachage `hash2`. Voici les étapes détaillées de cette fonction :

1. Initialisation du PRNG : Un générateur de nombres pseudo-aléatoires (PRNG) est initialisé avec le hachage `hash2` ;

2. Génération des défis : Pour chaque élément `i_star` (de 0 à $\tau - 1$), un nombre aléatoire est généré et utilisé pour calculer un indice de défi `i_star[l]` en prenant le modulo du nombre de parties (`N_PARTIES`).

Troisième partie

III/- Résultats expérimentaux

3 Bibliothèques utilisées

1. `gmp` pour la génération de nombres aléatoires,
2. `openssl` pour l'utilisation du hash Keccak.

Références

[1] Jean-Charles Faugère et al. « Cryptanalysis of MinRank ». In : Advances in Cryptology – CRYPTO 2008 (2008), p. 280-296. doi : https://doi.org/10.1007/978-3-540-85174-5_16.

[2] Yuval Ishai et al. Zero-Knowledge from Secure Multiparty Computation. url : https://web.cs.ucla.edu/~rafail/PUBLIC/77.pdf.

[3] Thibauld Feneuil. Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP. url : https://eprint.iacr.org/2022/1512.pdf.

[4] Thibauld Feneuil. Recent Advances in MPCitH-based Post-Quantum Signatures. url : https://www.thibauld-feneuil.fr/docs/talks/2024-03-22_Seminar-Rennes_tcith.pdf.

[5] Javier Verbel Gora ADJ Luis Rivera-Zamarripa. MiRitH (MinRank in the Head). url : http://www-cs-faculty.stanford.edu/~uno/abcde.html.