Schéma de signature post-quantique MiRitH

COUTE Maxime, MAGOIS Jules

18 février 2025

Plan de la présentation

- Présentation générale
  - Problème MinRank
  - MinRank in the Head (MiRitH)
  - paramètres de la signature
- Multi Party Computations
- Génération de la signature
- Vérification de la signature

Problème MinRank

$M_1, \ldots, M_m \in \mathcal{M}_n(\mathbb{K}), \quad r < n.$
Trouver $\alpha \in \mathbb{K}^m$ tel que :

$$rg\left(\sum_{i=1}^{m} \alpha_i M_i\right) \leq r$$

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 $M_\alpha = \left[M_\alpha^L | M_\alpha^R\right]$, $M_\alpha^L \in \mathbb{F}_q^{m\times(n-r)}$.

Problème MinRank

$M_1, \ldots, M_m \in \mathcal{M}_n(\mathbb{K}), \quad r < n.$
Trouver $\alpha \in \mathbb{K}^m$ tel que :

$$rg\left(\sum_{i=1}^{m} \alpha_i M_i\right) \leq r$$

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 $M_\alpha = \left[M_\alpha^L | M_\alpha^R\right]$, $M_\alpha^L \in \mathbb{F}_q^{m \times (n-r)}$.

MinRank in the Head

- effectue une preuve zero-knowledge (τ rounds)
  - vérifie une solution à une instance du problème MinRank
  - simule N parties (MPC)
  - la vérification est distribuée entre toutes les parties
- utilise l'heuristique de Fiat-Shamir
  - au lieu de générer les challenges aléatoirement, on utilise des hash

MinRank in the Head

- effectue une preuve zero-knowledge (τ rounds)
  - vérifie une solution à une instance du problème MinRank
  - simule N parties (MPC)
  - la vérification est distribuée entre toutes les parties
- utilise l'heuristique de Fiat-Shamir
  - au lieu de générer les challenges aléatoirement, on utilise des hash

Paramètres

Paramètres : $(\lambda, k, m, n, r, s, N, \tau)$

λ	sécurité
$k+1$	le nombre de matrices de chaque problème MinRank
(m, n)	la dimension de ces matrices
r	le rang de la matrice solution
(s, m)	la dimension du défi R
N	le nombre de parties
τ	le nombre de tours

Paramètres

Paramètres : $(\lambda, k, m, n, r, s, N, \tau)$
Probabilité pour chaque tour :

MinRank qu'une matrice aléatoire donne la solution : q^{-s}

ZKProof qu'un prouveur malhonnête convainque un vérifieur honnête : $1/N$

Prise en compte du nombre de tours : mise à la puissance τ
Taille de la clé $\simeq\ 2\lambda + \tau\big(A + \lambda B\big)$, où A dépend de la dimension des matrices, et $B = O(\log(N))$

On peut choisir différents paramètres :

- pour obtenir une clé petite (mais demandant beaucoup de calcul)
- pour faire peu de calculs (mais produisant une grande clé)

Paramètres

Paramètres : $(\lambda, k, m, n, r, s, N, \tau)$
Probabilité pour chaque tour :

- **MinRank** qu'une matrice aléatoire donne la solution : q^{-s}
- **ZKProof** qu'un prouveur malhonnête convainque un vérifieur honnête : $1/N$

Prise en compte du nombre de tours : mise à la puissance τ
Taille de la clé $\simeq\ 2\lambda + \tau\big(A + \lambda B\big)$, où A dépend de la dimension des matrices, et $B = O(\log(N))$

On peut choisir différents paramètres :

- pour obtenir une clé petite (mais demandant beaucoup de calcul)
- pour faire peu de calculs (mais produisant une grande clé)

Paramètres

Paramètres : $(\lambda, k, m, n, r, s, N, \tau)$
Probabilité pour chaque tour :

- MinRank qu'une matrice aléatoire donne la solution : q^{-s}
- ZKProof qu'un prouveur malhonnête convainque un vérifieur honnête : $1/N$

Prise en compte du nombre de tours : mise à la puissance τ
Taille de la clé $\simeq 2\lambda + \tau\big(A + \lambda B\big)$, où A dépend de la dimension des matrices, et $B = O(\log(N))$

On peut choisir différents paramètres :

- pour obtenir une clé petite (mais demandant beaucoup de calcul)
- pour faire peu de calculs (mais produisant une grande clé)

Paramètres

Paramètres : $(\lambda, k, m, n, r, s, N, \tau)$
Probabilité pour chaque tour :

- **MinRank** qu'une matrice aléatoire donne la solution : q^{-s}
- **ZKProof** qu'un prouveur malhonnête convainque un vérifieur honnête : $1/N$

Prise en compte du nombre de tours : mise à la puissance τ
Taille de la clé $\simeq 2\lambda + \tau\big(A + \lambda B\big)$, où A dépend de la dimension des matrices, et $B = O(\log(N))$

On peut choisir différents paramètres :

- pour obtenir une clé petite (mais demandant beaucoup de calcul)
- pour faire peu de calculs (mais produisant une grande clé)

Génération de la signature

- **Phase 1**
  - création d'une paire de clés (publique et secrète)
  - partage de α, A, K, C
  - génération des engagements
- **Phase 2** calcul de h_1, génération de matrices $R_1, \ldots, R_\tau$
- **Phase 3** chaque partie i du tour r calcule $S_{r,i}, V_{r,i}$
- **Phase 4**
  - calcul de h_2 à partir des $S_{r,i}, V_{r,i}$
  - génération des indices j_r
- **Phase 5** la signature est la concaténation de h_1, h_2 et pour chaque tour r :
  - les graines $s_{r,i}$ pour tout $i \neq j_r$
  - l'engagement de la partie j_r
  - la matrice S_{r,j_r}

Génération des partages

On commence par générer une matrice aléatoire A.
Pour chaque tour, et chaque partie (sauf la dernière), on génère les données suivantes :

- une graine pour le PRNG utilisée pour la suite
- une structure `PartyData` contenant les partages de α, A, C, K
- l'engagement, un hash dépendant de r, i et de la graine

Dernière partie : le calcul de α, A, K, C est différent, l'engagement dépend de ces matrices.

Génération des partages

On commence par générer une matrice aléatoire A.
Pour chaque tour, et chaque partie (sauf la dernière), on génère les données suivantes :

- une graine pour le PRNG utilisée pour la suite
- une structure `PartyData` contenant les partages de α, A, C, K
- l'engagement, un hash dépendant de r, i et de la graine

Dernière partie : le calcul de α, A, K, C est différent, l'engagement dépend de ces matrices.

Génération des partages

```
for (uint round = 0; round < params.tau; round++) {
  generate_seed(round_seed);
  TreePRG(&salt, &round_seed, /*...*/);
  for (uint party = 0; party < N - 1; party++) {
    // feed the PRG with the party seed
    PRG_init(&salt, &party_seed, lambda, &prg_state);
    // generate A, K, C...
    hash0(commits,  /*...*/, round, party, party_seed);
  }
  // last party: compute last alpha, K, C...
  hash0_last(commits, /*...*/ party_seed, alpha, K, C);
}
```

Premier Défi

Calcul de h_2, le hash de la concaténation des messages suivants :

- `msg` le message à signer
- `salt` une valeur aléatoire de taille 2λ
- `commits` la concaténation des engagements pour chaque tour et partie

Cette valeur est ensuite utilisée comme graine pour générer τ matrices $R_1, \ldots, R_\tau$

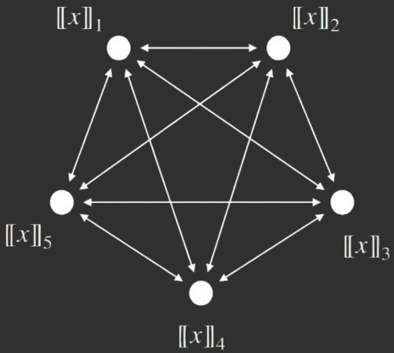
Premier Défi

```c
void phase_two(Matrix *challenges, uchar *h1, /*...*/ uchar
    ***commits) {
  // compute the hash
  hash1(h1, message, salt, /*...*/ commits);
  // generate challenges
  prg_first_challenge(challenges, h1);
}
```

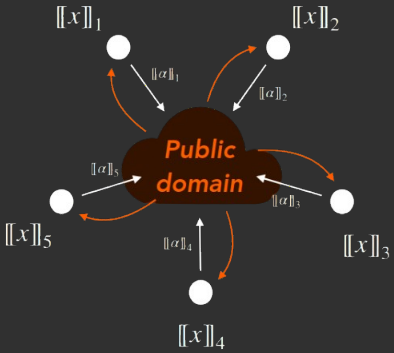
```c
void prg_first_challenge(Matrix *challenges, uchar *h1) {
  PRG_init(&h1, NULL, lambda, &prg_state);
  for (uint round = 0; round < tau; round++) {
    generate_random_matrix(&challenges[round], prg_state);
  }
  gmp_randclear(prg_state);
}
```

MPC

$$x = [[x]]_1 + [[x]]_2 + [[x]]_3 + [[x]]_4 + [[x]]_5$$



MPC-in-the-Head



Protocole MPC

On applique le protocole MPC pour chaque tour, avec les données générées précédemment :

- chaque partie
  - calcule $M = \sum_i \alpha_i M_i$ où α est généré à la phase 1
  - calcule $S = RM_{right} + A$ où R vient de la phase 2, A de la 1
- on somme tous les S_i obtenus pour obtenir $\tilde{S}$
- chaque partie calcule $V = \tilde{S}K - RM_{left} - C$, où K, C vient de la phase 1

Protocole MPC

```c
void phase_three(Matrix *challenges, /*...*/) {
  allocate_matrix(&S, GF_16, S_size);
  for (uint round = 0; round < params.tau; round++) {
    for (uint party = 0; party < N; party++) {
      compute_local_m(&parties[round][party], /*...*/);
      compute_local_s(&parties[round][party], /*...*/);
    }
    compute_global_s(&S, parties[round], N);
    for (uint party = 0; party < N; party++) {
      compute_local_v(&parties[round][party], S, /*...*/);
    }
  }
  clear_matrix(&S);
}
```

Deuxième défi

Un deuxième hash h_2 est calculé. C'est le haché (SHA3) de:

- le message `msg`
- la valeur aléatoire `salt`
- le premier hash h_1
- la représentation en `unsigned char`, pour chaque tour r et partie i, de $S_{r,i}$ et $V_{r,i}$

Cette valeur est utilisée pour générer τ indices $j_1, \ldots, j_\tau$: la partie j_r est celle qui n'est pas divulguée au tour r.

Deuxième défi

Un deuxième hash h_2 est calculé. C'est le haché (SHA3) de:

- le message `msg`
- la valeur aléatoire `salt`
- le premier hash h_1
- la représentation en `unsigned char`, pour chaque tour r et partie i, de $S_{r,i}$ et $V_{r,i}$

Cette valeur est utilisée pour générer τ indices $j_1, \ldots, j_\tau$: la partie j_r est celle qui n'est pas divulguée au tour r.

Deuxième défi

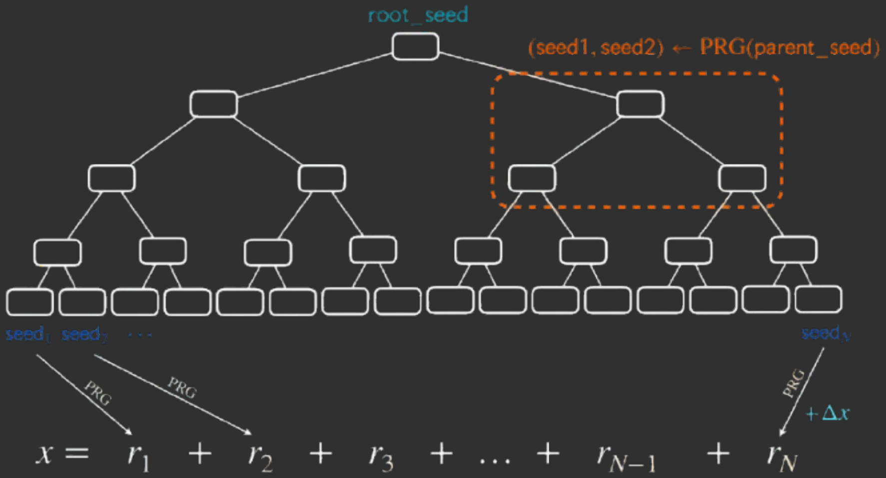
```c
void prg_second_challenge(/*...*/) {
  PRG_init(&h2, NULL, lambda, &prg_state);

  uchar *rand_bytes = malloc(sizeof(uchar) * 4 * tau);
  PRG_bytes(prg_state, 4 * tau, rand_bytes);

  for (uint round = 0; round < tau; round++) {
    uint i = 4 * round;
    challenges[round] = (uint)rand_bytes[i] << 24;
    challenges[round] += (uint)rand_bytes[i + 1] << 16;
    challenges[round] += (uint)rand_bytes[i + 2] << 8;
    challenges[round] += uint)rand_bytes[i + 3];
    challenges[round] %= N;
  }
}
```

Résultat

La signature est la concaténation de h_1, h_2 et pour chaque tour r :

- les graines $s_{r,i}$ pour tout $i \neq j_r$
- l'engagement de la partie j_r
- la matrice S associée à j_r

Seed Tree


root_seed
(seed1, seed2) ← PRG(parent_seed)
PRG
PRG
PRG
$+\Delta x$
$x = r_1 + r_2 + r_3 + \ldots + r_{N-1} + r_N$

```c
// Function to generate bytes from the PRG state
void PRG_bytes(gmp_randstate_t prg_state, size_t length, unsigned char *output) {
  // the GMP integer holding the random byte
  mpz_t temp;
  mpz_init(temp);

  for (uint i = 0; i < length; i++) {
    // generate a random byte
    mpz_urandomb(temp, prg_state, 8);
    int8_t random_byte = mpz_get_ui(temp);
    output[i] = (uchar)random_byte;
  }
  mpz_clear(temp);
}
```

Bibliographie

[1] MiRitH (MinRank in the Head), Javier Verbel Gora ADJ Luis Rivera-Zamarripa.

[2] Zero-Knowledge from Secure Multiparty Computation, Yuval Ishai et al.

[3] Building MPCitH-based Signatures from MQ, MinRank, Rank SD and PKP, Thibauld Feneuil.

[4] Recent Advances in MPCitH-based Post-Quantum Signatures, Thibauld Feneuil.

[5] Cryptanalysis of MinRank, Jean-Charles Faugère et al.