# TP initiation à la cryptographie post-quantique

Jean-Christophe Deneuville
jean-christophe.deneuville@enac.fr

15 décembre 2020

## Contexte

Comme nous l'avons vu en cours, le chiffrement fondé sur codes correcteurs d'erreurs souffre généralement de tailles de clés publiques importantes (plusieurs kilo voire méga octets). Afin de garder un niveau de praticité raisonnable, il est possible d'utiliser deux leviers :

1. l'utilisation de codes structurés pour réduire significativement la taille de la clé publique,
2. un échange de clés, suivi d'un chiffrement symétrique (on parle de chiffrement hybride, ou de KEM-DEM, pour Key Encapsulation Mechanism & Data Encryption Mechanism).

L'objectif de ce TP est de réaliser une preuve de concept d'un algorithme d'échange de clés basé sur les codes correcteurs d'erreurs quasi-cyclique, nommé Ouroboros [1].

Tous les exemples de codes sont donnés en Python3, libre à vous de les adapter dans un autre langage si vous préférez.

> **Attetion :** cette implémentation est/sera trivialement non sécurisée et ne doit pas être utilisée dans un contexte utilisant des données sensibles.
>
> *Don't roll your own crypto!*

## Matériel nécessaire

Commencez par télécharger l'archive à l'adresse suivante : https://filesender.renater.fr/?s=download&token=9f729345-d9ff-4518-ace7-82e48160d76b

Elle contient le squelette des routines nécessaires à l'implémentation.

## 1 Pré-requis : codes QC-MDPC et leur décodage

Avant de comprendre comment fonctionne le protocole d'échange de clés Ouroboros, nous avons besoin de définir ce qu'est un code QC-MDPC et comment on peut décoder de tels codes efficacement.

Dans la suite, nous travaillerons toujours (sauf mention explicite du contraire) modulo 2 (dans $\mathbb{F}_2$ donc). Les vecteurs sont notés en gras en minuscule, les matrices en gras en majuscules.

### 1.1 Codes QC-MDPC

QC-MDPC code signifie Quasi-Cyclic Moderate Density Parity Check code, ou en français un code quasi-cyclique à matrice de parité modéremment creuse. Il s'agit d'une famille de codes admettant une matrice de parité de la forme suivante :

$$\mathbf{H} = \begin{pmatrix} \mathbf{h}_0 & \mathbf{h}_1 \\ \circlearrowleft & \circlearrowleft \end{pmatrix}, \tag{1}$$

avec $\mathbf{h}_0, \mathbf{h}_1 \in \mathbb{F}_2^n$ de "faible" poids de Hamming $w < \sqrt{n}$. La notation $\circlearrowleft$ signifie que chaque fois que l'on descend d'une ligne dans la matrice, $\mathbf{h}_i$ est décalé circulairement d'une position vers la droite.

### 1.2 Algorithme de Bit-Flipping étendu

En 1962, Gallagher a introduit l'algorithme du bit flipping pour décoder itérativement des code LDPC (L signifiant Low). L'idée est — en calculant le syndrome du mot reçu — de regarder quelles sont les équations de parité non satisfaites, et de flipper les bits du mot reçu impliquant le plus de telles équations, afin d'en réduire un maximum à chaque itération.

Plus formellement, si $\mathbf{v} = \mathbf{m} + \mathbf{e}$ est le mot reçu avec $\mathbf{m}$ un mot du code et $\mathbf{e}$ une erreur de petit poids, alors en calculant $\mathbf{Hv} = \mathbf{H}(\mathbf{m} + \mathbf{e}) = \mathbf{He}$, on obtient une série d'équations de parité insatisfaites. En regardant les bits de $\mathbf{e}$

impliquant le plus d'équations de parité non-satisfaites, il est possible, par maximum de vraisemblance, de retrouver les bits en erreurs dans le mot reçu, et donc de les corriger.

Ce faisant, l'algorithme de bit flipping de Gallagher permet de modifier le mot reçu jusqu'à l'obtention d'un syndrome nul, témoignant d'un décodage complet, menant au recouvrement du mot initialement émis.

Pour Ouroboros, nous aurons besoin d'une version étendue tolérant une erreur supplémentaire. Autrement dit, on ne cherchera pas à ramener le nombre d'équations de parité non-satisfaites à zéro, mais à un seuil acceptable. Cet algorithme étendu est décrit ci-dessous.

Plus formellement, si $\mathbf{v} = \mathbf{m} + \mathbf{a} + \mathbf{b}$ est le mot reçu avec $\mathbf{m}$ un mot du code, et $\mathbf{a}$ et $\mathbf{b}$ des erreurs de petit poids, l'objectif du bit flipping étendu sera de corriger l'erreur $\mathbf{a}$ avec comme marge d'erreur le poids du vecteur $\mathbf{b}$.

**Remarque :** Le schéma a été conçu de telle sorte que cette marge d'erreur ne puisse pas (avec une probabilité super-polynomialement proche de 1) conduire à un mauvais décodage.

**Algorithm 1:** extended-Bit-Flipping$(\mathbf{x}, \mathbf{y}, \mathbf{s}, t, w, w_e)$

**Input:** $\mathbf{x}, \mathbf{y}$, and $\mathbf{s} = \mathbf{x}\mathbf{r}_2 + \mathbf{y}\mathbf{r}_1 + \boldsymbol{e}$, threshold $t$ required to flip a bit, weight $w$ (resp. $w_e$) of $\mathbf{r}_1$ and $\mathbf{r}_2$ (resp. $\boldsymbol{e}$).
**Output:** $(\mathbf{r}_1, \mathbf{r}_2)$ if the algorithm succeeds, $\perp$ otherwise.

$(\mathbf{u}, \mathbf{v}) \leftarrow (\mathbf{0}, \mathbf{0}) \in (\mathbb{F}_2^n)^2$, $\mathbf{H} \leftarrow \left(\mathbf{rot}(\mathbf{y})^\top, \mathbf{rot}(\mathbf{x})^\top\right) \in \mathbb{F}_2^{n \times 2n}$, syndrome $\leftarrow \mathbf{s}$;
**while** $[wt(\mathbf{u}) \neq w$ **or** $wt(\mathbf{v}) \neq w]$ **and** $wt(\text{syndrome}) > w_e$ **do**
  sum $\leftarrow$ syndrome$\times\mathbf{H}$; `/* No modular reduction */`
  flipped_positions $\leftarrow \mathbf{0} \in \mathbb{F}_2^{2n}$;
  **for** $i \in [\![0, 2n-1]\!]$ **do**
    **if** sum$[i] \geq t$ **then**
      flipped_positions$[i]$ = flipped_positions$[i] \oplus 1$;
  $(\mathbf{u}, \mathbf{v}) = (\mathbf{u}, \mathbf{v}) \oplus$ flipped_positions;
  syndrome = syndrome $-\mathbf{H} \times$ flipped_positions$^\top$;
**if** $wt\left(\mathbf{e_c} - \mathbf{H} \times (\mathbf{u}, \mathbf{v})^\top\right) > w_{\boldsymbol{e}}$ **then**
  **return** $\perp$;
**else**
  **return** $(\mathbf{u}, \mathbf{v})$;

## 2 Présentation du protocole d'échange de clés Ouroboros

Dans le protocole d'échange de clés Ouroboros, Alice génère un code quasi-cyclique de matrice de parité aléatoire $(\mathbf{1}, \mathbf{h})$, et calcule le syndrome $\mathbf{s}$ d'un mot de petit poids $(\mathbf{x}, \mathbf{y})$ pour obtenir sa clé publique.

Bob génère une clé aléatoire $\boldsymbol{\epsilon}$ et l'encapsule comme décrit dans la figure 1.

Alice peut alors retrouver cette clé en enlevant un grand nombre d'erreurs à l'aide de sa clé secrète, puis en appliquant l'algorithme de décodage.

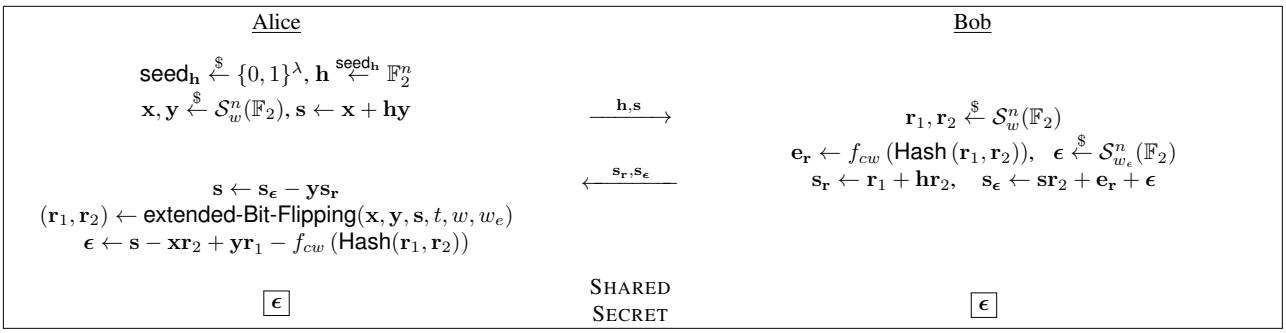


FIGURE 1 – Description du protocole Ouroboros.

**Exemple de paramètres.** Pour 80 bits de sécurité, on considèrera les paramètres suivants : $n = 5851, w = 47, w_e = 94, t = 30$.

## 3 Implémentation des routines

Nous allons maintenant passer à la partie pratique : l'implémentation. Nous aurons besoin des routines suivantes :

— calcul du poids d'un vecteur

— génération d'un mot aléatoire de longueur $n$ et de poids $w$
— multiplication de 2 vecteurs
— produit vecteur-matrice, sans réduction modulo 2, pour le calcul des équations de parité
— produit matrice-vecteur, avec réduction modulo 2, pour le calcul de syndrome
— addition de vecteurs
— fonction de hachage en poids faible
— génération de la matrice quasi-cyclique associée à la clé secrète.

Nous travaillerons exclusivement avec des listes pour les vecteurs, et des listes de listes pour les matrices.

## 3.1 Calcul du poids d'un vecteur

En utilisant la méthode `count`, écrire la fonction `wt(v)` qui prend en entrée un vecteur `v` et retourne son poids de Hamming.

## 3.2 Génération d'un mot aléatoire de petit poids

En utilisant la fonction `shuffle` de la librairie `random`, écrire une fonction `randomVectOfFixedWeight(w,n)` qui prend en entrée la longueur $n$ du vecteur à créer ainsi que son poids de Hamming `w`, génère un vecteur de `w` coordonnées à 1 et `n-w` coordonnées à 0 et qui lui applique une permutation aléatoire.

## 3.3 Multiplication de 2 vecteurs

La multiplication de deux vecteurs `u` et `v` s'effectue, comme vu en cours, comme une multiplication polynomiale dans l'anneau $\mathbb{F}_2[x]/\left(x^n-1\right)$. C'est à dire que $\mathbf{u}\times\mathbf{v}=\mathbf{w}$ avec :

$$w_k = \sum_{i+j=k} u_i v_j, \text{ pour } 0 \leq k \leq n-1.$$

Implémenter cette fonction.

Astuce : Afin d'optimiser l'éxécution, vous pourrez faire l'hypothèse que le premier argument est potentiellement un vecteur creu (beaucoup de $O$), et ne procéder à la boucle interne que si la coordonnée courante de ce vecteur est non-nulle.

## 3.4 Produit vecteur-matrice sans réduction modulo 2

Pour `u` vu comme un vecteur $\mathbf{u}\in\mathbb{Z}^n$ et `M` vue comme une matrice $\mathbf{M}\in\mathbb{Z}^{n\times m}$ (à $n$ lignes et $m$ colonnes), le produit $\mathbf{v}=\mathbf{u}\mathbf{M}$ donnera une liste `v` à $m$ coordonnées. Implémenter cette fonction. Attention à ne pas appliquer de réduction modulo 2 !

## 3.5 Produit matrice-vecteur avec réduction modulo 2

Pour `v` vu comme un vecteur $\mathbf{v}\in\mathbb{F}_2^m$ et `M` vue comme une matrice $\mathbf{M}\in\mathbb{Z}^{n\times m}$ (à $n$ lignes et $m$ colonnes), le produit $\mathbf{u}=\mathbf{M}\mathbf{v}$ donnera une liste `v` à $m$ coordonnées. Implémenter cette fonction. Cette fois-ci, le résultat devra être réduit modulo 2 !

## 3.6 Addition de vecteurs

Écrire une fonction qui étant données deux liste `u` et `v` de même longueur retourne leur somme.

## 3.7 Fonction de hachage en poids faible

Ouroboros nécessite une fonction de hachage qui retourne des mots de poids faible. Nous utiliserons la librairie `hashlib` pour accéder à SHA-512, nous considérerons la sortie $h$ de cette fonction comme un entier de 512 bits, dont nous nous servirons pour déterminer les quelques positions à 1 de notre vecteur résultat.

Cette fonction est fournie dans l'archive téléchargée.

### 3.8 Génération de la matrice quasi-cyclique

Dans la ligne 1 de l'algorithme du bit flipping étendu, on génère la matrice $\mathbf{H}$ comme suit :

$$\mathbf{H} = \left(\mathbf{rot}(\mathbf{y})^\top, \mathbf{rot}(\mathbf{x})^\top\right) = \left(\begin{array}{cccc|cccc} y_0 & y_{n-1} & \cdots & y_1 & x_0 & x_{n-1} & \cdots & x_1 \\ y_1 & y_0 & \cdots & y_2 & x_1 & x_0 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ y_{n-1} & y_{n-2} & \cdots & y_0 & x_{n-1} & x_{n-2} & \cdots & x_0 \end{array}\right) \tag{2}$$

Écrire la fonction `generateH(x,y)` qui étant donnés $\mathbf{x}$ et $\mathbf{y}$, génère la matrice $\mathbf{H}$ comme décrit dans l'équation (2).

# 4 Implémentation de l'algorithme Ouroboros

## 4.1 Génération de la clé

Implémenter la fonction `keygen(params)` qui étant donnés les paramètres $n, w, w_e$ et $t$ retourne une clé privée `sk = (x, y)` et une clé publique `pk = (h, s)` tel que décrit dans la figure 1.

## 4.2 Encapsulation de la clé partagée

De même, créer une fonction `encapsulate`, regroupant toutes les opérations réalisées par Bob dans la figure 1.

## 4.3 Décapsulation de la clé

Enfin, écrire la fonction `decapsulate` qui réalise les dernières opérations de la partie d'Alice dans la figure 1.

## 4.4 Vérification de l'exactitude de l'implémentation

Nous allons maintenant nous assurer que la clé décapsulée par Alice est bien celle qui avait été encapsulée par Bob. Pour ce faire, commencez par (artificiellement) rajouter la clé encapsulée en sortie de l'algorithme `encapsulate`. En fin d'algorithme, comparez la clé qui est retrouvée après algo de bit flipping étendu à celle précédemment obtenue. Sont-elles égales ?

# Références

[1] Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Ouroboros : A simple, secure and efficient key exchange protocol based on coding theory. In *International Workshop on Post-Quantum Cryptography*, pages 18–34. Springer, 2017.