



Documentation technique

ML-KEM

Réalisé par :

- Louis Landouzy
- Hamza Berbaché
- Benoît Wattinne
- Clément Gorse
- Enzo Barea
- Aurélien Degain
- Rémi Protin

Supervisé par :

- Gabriel Chênevert

Table des matières

A.	Introduction	3
B.	Choix techniques	4
1.	Language.....	4
2.	Types.....	4
3.	Classes	5
C.	Dépendances	10
1.	BitsToBytes.....	10
2.	BytesToBits.....	11
3.	ByteEncode	11
4.	ByteDecode.....	12
5.	SampleNTT.....	12
6.	SamplePolyCBD η	13
7.	NTT	14
8.	NTT-1.....	15
9.	BitRev7.....	16
10.	findNumBits	16
11.	findSmallestPrimitiveRoot	16
12.	Zeta	16
13.	Surcharge d'opérateur * NTT	17
D.	K-PKE	18
14.	K-PKE.KeyGen.....	18
15.	K-PKE.Encrypt.....	18
E.	Améliorations	19
F.	Conclusion	20
G.	Références	21

A. Introduction

Suite au premier rapport sur ML-KEM qui exposait le principe de ce nouveau standard cryptographique, ce document s'inscrit dans le cadre de la réponse au besoin pressant de défenses contre la menace croissante des ordinateurs quantiques. Ces derniers, avec leur capacité à compromettre les protocoles classiques, ont incité le NIST à lancer une campagne pour l'élaboration de nouvelles normes, dont la complexité mathématique défierait tant les ordinateurs quantiques que les ordinateurs classiques. Notre objectif ici est de prendre en main la documentation actuelle, encore à l'état de « brouillon », pour procéder à l'implémentation du protocole.

Dans cette optique, ce document technique expose l'implémentation de l'algorithme ML-KEM. Il débutera par l'exposition des choix techniques et fonctionnels ayant orienté le développement, puis il abordera les fonctions principales pour établir les bases de l'algorithme. Enfin, nous détaillerons l'implémentation des algorithmes de ML-KEM.

En parallèle de cette exploration technique, ce projet a été mené en utilisant des méthodes de gestion agile. L'utilisation de l'outil de gestion de projet Trello a permis de structurer et de suivre les étapes clé du processus. Des comptes-rendus réguliers ont été établis à chaque réunion, assurant une traçabilité et une gestion efficace des tâches. Chaque contributeur s'est vu attribuer une tâche suite à des discussions visant à définir la direction à prendre pour mener à bien le projet.

Nous souhaitons exprimer notre gratitude envers M. Chênevert, notre enseignant et responsable de ce projet, pour son précieux accompagnement et ses conseils éclairés tout au long de ce projet. Sa contribution a été fondamentale pour faire avancer notre cheminement et guider le développement du projet.

B. Choix techniques

1. Language

Le choix du langage s'est avéré être une décision cruciale. Python a été envisagé pour sa polyvalence, sa syntaxe claire, et la maîtrise qu'en avait le groupe. Une option aurait été d'intégrer le code dans un notebook pour une approche plus didactique. Cependant, notre objectif était de développer une implémentation robuste et hautement modulaire, permettant de distinguer clairement les différentes entités impliquées dans la conception de l'algorithme. C'est pourquoi nous avons opté pour un langage orienté objet avec un typage fort. Le choix du C++ s'est rapidement imposé pour sa performance élevée, sa gestion efficace de la mémoire (critique dans les applications à haute performance), ainsi que sa capacité à créer des structures modulaires et hautement optimisées. Le C++ est largement utilisé dans des domaines où les performances sont primordiales, tels que les systèmes embarqués et les logiciels de communication. Toutefois, il convient de noter que le C++ peut être plus complexe à apprendre et à utiliser que Python, et que le développement initial peut nécessiter davantage de temps et d'efforts. Bien que le typage statique puisse initialement ralentir le développement, il peut également conduire à une meilleure performance et une fiabilité accrue à long terme. Ces défis ont été rencontrés au fur et à mesure de l'avancement du projet. Avec le recul, il est évident que, d'un point de vue purement didactique, le choix de Python aurait été plus approprié.

Maintenant que les fondations sont en place et bien structurées, le code source est hautement lisible, ce qui facilite sa maintenance et son développement ultérieur.

2. Types

Dans cette implémentation, les bits manipulés sont représentés par des booléens

Les autres variables, comme les coefficients ou les octets, sont représentées par des types non signés. Ce choix a été fait car nous manipulons parfois les données sous forme de bits, et nous n'utilisons jamais de nombres négatifs, que ce soit pour les coefficients des NTT et des polynômes, ou pour les indices et autres représentations binaires.

Ainsi, les octets sont représentés par des entiers non signés de 8 bits :

uint8_t

Les coefficients sont représentés sur 16 bits. En effet, leur valeur ne pouvant pas dépasser $q = 3329$, soit 110100000001 en binaire, ils sont donc codés sur 12 bits au maximum :

uint16_t

3. Classes

Chaque entité ou type de données a été représenté par des classes. Cette approche permet de définir de manière précise chaque donnée et évite toute confusion dans l'implémentation. Par exemple, les coefficients de la NTT et des polynômes, bien que tous deux des entiers non signés sur 16 bits, sont distincts dans notre structure. Un échange malencontreux entre ces entités ne générerait aucune erreur de compilation, mais pourrait entraîner un échec calculatoire lors du déroulement du protocole. De plus, l'utilisation de classes simplifie l'écriture du code grâce à la possibilité de surcharger les opérateurs. Par exemple, il est plus compréhensible de multiplier deux matrices de NTT ensemble de la manière suivante :

*MatriceNTT result = matrice1 * matrice2*

Toute la complexité de cette opération est encapsulée dans la surcharge de l'opérateur *, ce qui rend l'écriture du code plus claire et plus concise.

```
MatriceNTT operator * (NTTmatrice m){  
    //Opération matricielle  
}
```

Cette approche rend le code plus maintenable car toutes les opérations et fonctions concernant une entité sont présentes dans sa classe respective. De plus, pour chaque classe, il est possible de surcharger les opérateurs de flux, facilitant ainsi leur rendu et leur affichage.

Au niveau le plus bas, des bits et des octets sont manipulés à l'aide de différentes fonctions, notamment pour les conversions et l'encodage. Tout d'abord, une classe **Bit** est définie pour représenter un seul bit, qui est un booléen. Ainsi, même les entités les plus simples sont représentées par des classes pour maintenir le principe de modularité.

```
class Bit {  
    private:  
        bool bit  
    public:  
        // Constructeurs  
        // Destructeurs  
        // Getters / Setters  
        // Méthodes utiles  
}
```

Ensuite, il y a la classe **BitArray**, qui est l'un des éléments principaux des fonctions de manipulation de données. Il s'agit d'un tableau de **Bit**, ici représenté par un vecteur et ses méthodes associées, dont la méthode `bytesToBits` qui sera présentée par la suite.

```
class BitArray {
private:
    vector<Bit> bits
    int size
public:
    // Méthodes usuelles ...

    // Méthode spéciale
    BitArray bytesToBits(ByteArray b)

    // Surcharge de l'opérateur <<
    friend ostream& operator<<(ostream& os, BitArray b)
}
```

À un niveau supérieur, les octets sont représentés par la classe **Byte** de type `uint8_t`

```
class Byte {
private:
    uint8_t byte
}
```

De la même manière que pour **Bit**, il existe une classe **ByteArray** qui fait également partie des éléments de manipulation de données. Il s'agit d'un tableau de **Byte** qui possède également des méthodes importantes telles que [bitsToBytes](#), [byteEncode](#) et [byteDecode](#), qui seront discutées dans la partie **Dépendances**.

```
class ByteArray {
private:
    vector<Byte> bytes
    int size
public:

    ByteArray bitsToBytes(BitArray b)
    ByteArray byteEncode(IntArray F
    IntArray byteDecode(ByteArray B)
}
```

Il est ainsi possible de passer d'une entité à l'autre en utilisant les méthodes de conversion.

Il s'agit maintenant d'aborder les entités qui constituent la base du problème mathématique derrière ML-KEM : les polynômes et leur représentation dans le domaine NTT.

Premièrement, les polynômes sont des tableaux de $n = 256$ coefficients de nombres entiers modulo q , soit $\mathbb{Z}_q^{256}$. Il a été choisi de représenter les coefficients sous forme de classe afin de simplifier et de contrôler les calculs entre les coefficients de polynômes. Cela garantit notamment que les calculs sont toujours effectués modulo q .

```
class PolyCoef {  
  
    private:  
        uint16_t value;  
  
    public:  
        PolyCoef& operator+=(const PolyCoef& other) {  
            value = (value + other.value) % q;  
            return *this;  
        }  
  
        explicit operator uint16_t() const {  
            return value;  
        }  
}
```

Cette classe contient un unique attribut : un entier sur 16 bits représentant le coefficient modulo q . Les surcharges d'opérateurs simplifient l'écriture des calculs dans les fonctions qui les utilisent, sans nécessiter de recodage pour chaque cas (multiplication, addition, entre coefficient et scalaire, ou entre deux coefficients). Par exemple, la surcharge de l'opérateur `+=` permet de calculer la somme de deux **PolyCoef** et d'affecter le résultat à l'opérande de gauche.

Il est important de noter qu'il existe un opérateur de conversion permettant de passer d'un **PolyCoef** à un entier, mais cela nécessite une attention particulière pour éviter les erreurs invisibles. Par exemple, lors d'un calcul entre un **PolyCoef** et un entier, le **PolyCoef** est converti en entier, puis une somme d'entiers est calculée. Cependant, le modulo du calcul n'est pas contrôlé, ce qui peut entraîner des résultats erronés. Cet exemple ne tient pas compte de réalité mais envisage des limites à cette implémentation et pousse à plus de réflexion sur son utilité.

Cette classe implémente d'autres opérateurs, pour les calculs, même s'ils n'ont pas été identifiés dans la norme, pour anticiper de besoins et améliorer la robustesse de la classe.

PolyCoef est utilisé dans les polynômes. C'est ainsi que la classe **Poly** a été définie comme étant un tableau de **PolyCoef** de taille **n = 256**

```
class Poly{
private:
    std::array<NTTCoef, n> coefficients;

public:
    Poly(const NTT& f_hat);
    static Poly SamplePolyCBD_eta(const ByteArray& B, uint32_t eta);
    Poly& operator=(const Poly& other);
    Poly& operator+(const Poly& other);
};
```

Le constructeur de cette classe est en réalité la fonction NTT^{-1} , qui permet de convertir un polynôme de R_q vers un polynôme de T_q . Elle prend donc en entrée une instance de la classe **NTT**. De plus, seule l'addition peut-être surchargée et pas la multiplication car le but est d'utiliser la représentation NTT pour effectuer cette tâche.

Ensuite, il y a la classe **PolyMatrice**, qui représente les matrices et vecteurs de polynôme de taille $k \times k$ utilisées dans les fonctions de chiffrement.

```
class PolyMatrice{
private:
    std::vector<std::vector<Poly>> matrice;

public:
    void set(int i, int j, Poly value);
    Poly get(int i, int j);
    PolyMatrice operator+(PolyMatrice &m);
};
```

Les méthodes importantes sont `get` et `set` qui permettent d'ajouter des valeurs à des coordonnées précises de la matrice, comme c'est souvent le cas d'usage principal dans les algorithmes K-PKE. De même, des matrice sont parfois additionnées, ce qui a mené à l'ajout d'une surcharge de l'addition.

La représentation des polynômes dans le domaine de la NTT suis exactement le même schéma que les polynômes, ce sont 256 coefficient, toujours modulo q qui sont stockés dans un tableau. Ainsi il y a tout d'abord la classe **NTTCoef** :

```
class NTTCoef {

private:
    uint16_t value;

public:
    // Surcharges d'opérateur toujours modulo q
```

Il y a ensuite la classe **NTT**, qui cette fois-ci implémente la surcharge de la multiplication, qui correspond à la fonction *MultiplyNTTs* qui elle-même utilise la fonction *BaseCaseMultiply*. De même que pour Poly, le constructeur correspond à la fonction NTT et transforme un polynôme dans le domaine de la NTT.

```
class NTT{
    private:
        std::array<NTTCoef, n> coefficients;
        NTT MultiplyNTTs(const NTT& f, const NTT& g);
        void BaseCaseMultiply();

    public:
        NTT(const Poly& t);
        NTT& operator*(const NTT& other);
        static void testNTT();
};
```

Finalement, **NTTmatrice** correspond à une matrice ou un vecteur de **NTT** qui ici peut aussi multiplier des matrices entre elles.

```
class NTTmatrice{
    private:
        std::vector<std::vector<NTT>> matrice;

    public:
        NTTmatrice operator*(NTTmatrice &m);
};
```

C. Dépendances

Dans cette section, nous exposons les algorithmes auxiliaires tels que définis par le NIST pour la réalisation des schémas K-PKE et ML-KEM. Ces algorithmes ont été intégrés aux classes correspondant au type de données manipulées.

Nous avons également ajouté certaines méthodes en complément des spécifications lorsque nous avons jugé leur nécessité ou lorsque leur utilisation était sous-entendue dans la documentation.

1. BitsToBytes

La fonction BitsToBytes de la classe **ByteArray** a pour objectif de convertir un **BitArray** (représentant un vecteur de bits) en un **ByteArray** (représentant un vecteur d'octets). Cette transformation est fondamentale pour passer de la forme binaire, adaptée au traitement informatique, à la forme octet, plus appropriée pour le stockage et la transmission des données.

La fonction commence par calculer la taille du **ByteArray** de sortie en divisant la taille du **BitArray** d'entrée par 8, puisqu'un octet contient 8 bits. Ensuite, elle crée un nouveau **ByteArray** de la taille calculée précédemment. Elle parcourt ensuite chacun des bits du **BitArray** d'entrée à l'aide d'une boucle for. Pour chaque bit :

- Elle détermine l'index de l'octet correspondant dans le **ByteArray** de sortie, en divisant l'index du bit par 8.
- Elle calcule également l'index du bit au sein de cet octet, en prenant le reste de la division de l'index du bit par 8.
- Elle ajoute alors la valeur du bit (0 ou 1 décalé de (7 - *bitIndex*) positions vers la gauche) à l'octet correspondant dans le **ByteArray** de sortie, à l'aide de la méthode *add*.
- Une fois la boucle terminée, la fonction retourne le **ByteArray** résultant.

Cette implémentation permet donc de transformer efficacement un vecteur de bits en un vecteur d'octets, en plaçant chaque bit à la position appropriée au sein des différents octets du **ByteArray** de sortie.

Dans notre exemple, nous pouvons voir que notre **ByteArray** obtenu est bien le même que celui espéré, calculé à partir d'un **BitArray**.

```
BitArray : 0-1-0-1-0-1-0-1
----- BitsToBytes -----
Expected byte array from bit array : 01010101
ByteArray : 01010101
```

2. BytesToBits

La fonction `byteToBits` de la classe **BitArray** fait l'inverse de la fonction `bitsToBytes`. Elle a pour objectif de convertir un **ByteArray** (représentant un vecteur d'octets) en un **BitArray** (représentant un vecteur de bits).

La fonction `bytesToBits`, dans un premier temps, calcule la taille du **BitArray** de sortie en multipliant la taille du **ByteArray** d'entrée par 8, puisqu'un octet contient 8 bits. Elle crée ensuite un nouveau **BitArray** de la taille calculée précédemment. Enfin, elle parcourt chacun des octets du **ByteArray** d'entrée à l'aide d'une boucle `for`. Pour chaque octet :

- Elle extrait chacun des 8 bits de l'octet en les décalant de $(7 - j)$ positions vers la droite, puis en masquant le résultat avec 1 pour ne garder que le bit de poids faible.
- Elle définit alors la valeur du bit correspondant dans le **BitArray** de sortie à l'aide de la méthode `setBitIndex`.

Une fois la boucle terminée, le **BitArray** résultant est retourné. En reprenant la sortie de l'exemple précédents, nous obtenons bien le **BitArray** du début nous montrant bien que la fonction fonctionne correctement.

```
ByteArray : 01010101
----- BytesToBits -----
Expected bit array : 0-1-0-1-0-1-0-1
BitArray : 0-1-0-1-0-1-0-1
```

3. ByteEncode

La fonction `byteEncode` de la classe **ByteArray** permet de transformer un **IntArray** (représentant un vecteur de nombres) en un **ByteArray** (représentant un vecteur d'octets). Cela permet de sérialiser un vecteur de nombres en un vecteur d'octets, en décidant du nombre de bits sur lesquels chaque nombre sera codé.

Cette fonction est définie comme statique pour éviter d'avoir à initialiser un objet **ByteArray** en plus de celui qui sera retourné par la fonction.

Elle prend en entrée un tableau d'entiers et un entier **d**, représentant le nombre de bits sur lesquels chaque nombre sera codé. La fonction crée un **BitArray** et un **ByteArray** de taille égale au produit de la taille du tableau d'entiers et de **d**. Ensuite, elle itère à travers chaque élément du **IntArray**. Pour chaque nombre, elle parcourt les **d** bits de ce nombre. Pour chaque bit, elle les décale vers la droite de **d-1-j** pour placer le bit de poids fort à gauche, puis les stocke dans le **BitArray** avant de le convertir en **ByteArray** à l'aide de la fonction `bitsToBytes`.

Cet exemple montre un **IntArray** encodé sur 12 bits puis sur 8 bits pour lequel il est aisé de se convaincre du résultat.

```

IntArray : 15 3

----- Byte encode 12 (from IntArray) -----
Expected byte array : 00000000 11110000 00000011
ByteArray : 00000000 11110000 00000011

----- Byte encode 8 (from IntArray) -----
Expected byte array : 00001111 00000011
ByteArray : 00001111 00000011

```

4. ByteDecode

La fonction `byteDecode` de la classe **ByteArray** effectue l'opération inverse de la fonction `byteEncode`, c'est-à-dire qu'elle transforme un **ByteArray** (représentant un vecteur d'octets) en un **IntArray** (représentant un vecteur de nombres). Cela permet de désérialiser un vecteur d'octets en un vecteur de nombres, en prenant en compte le nombre de bits sur lesquels chaque nombre a été codé.

Cette fonction est également définie comme statique pour éviter d'avoir à initialiser un objet **ByteArray** en plus de celui qui sera retourné par la fonction. Elle prend en entrée un objet **ByteArray** et un entier **d**, représentant le nombre de bits sur lesquels chaque nombre a été codé. La fonction commence par convertir le **ByteArray** en **BitArray** à l'aide de la fonction `bytesToBits`. Ensuite, elle crée un **IntArray** de taille égale à la taille du **ByteArray** multipliée par 8 et divisée par **d** afin de stocker les valeurs décodées. En parcourant les éléments du **BitArray**, la fonction extrait **d** bits consécutifs de chaque élément, les convertit en une valeur entière, puis les stocke dans le **IntArray** qui est ensuite retourné.

Cet exemple qui reprend le résultat de l'encode précédent pour le décoder et parvient au **IntArray** initial.

```

ByteArray : 00001111 00000011

----- Byte decode (from ByteArray) -----
Expected int array : 15 3
IntArray : 15 3

```

5. SampleNTT

La fonction `sampleNTT` de la classe **NTT** vise à générer de manière pseudo-aléatoire un élément de l'anneau NTT en utilisant un générateur de nombres pseudo-aléatoires cryptographique appelé **XOF**.

Le déroulement de la fonction est le suivant :

- Un objet **NTT** est créé pour contenir tous les coefficients générés.
- Deux variables **i** et **j** sont initialisées à 0 pour servir d'itérateurs pour parcourir les coefficients.
- Tant que **j** est inférieur à 256 (la taille du polynôme NTT), la fonction effectue les étapes suivantes :
 - Elle extrait deux entiers **d1** et **d2** à partir du flux de données du générateur **XOF**

- **d1** est obtenu en concaténant le byte courant `XOF.getLastThree[i]` avec les 4 bits de poids faible du byte suivant `XOF.getLastThree[i+1]`.
- **d2** est obtenu en concaténant les 4 bits de poids fort de `XOF.getLastThree[i+1]` avec le byte `XOF.getLastThree[i+2]`.
- Si **d1** est strictement inférieur à **q** (le module), alors il est ajouté à l'index **j** du tableau de coefficients **a**.
- Si **d2** est strictement inférieur à **q** et que **j** est toujours inférieur à 256, alors il est ajouté à l'index **j** du tableau de coefficients **a**.
- Les indices **i** et **j** sont incrémentés.

En fin de compte, l'objet **NTT** contenant les 256 coefficients générés est retourné.

Cette fonction joue un rôle crucial dans la génération de polynômes aléatoires. Dans le test suivant, nous observons que nous avons réussi à générer deux fois, deux vecteurs de 256 éléments différents, ce qui démontre que les éléments sont générés de manière aléatoire.

```
Coefficients de l'élément NTT g0n0r0 :
713 3054 2897 2628 2934 2130 2233 2345 1366 2301 2413 1968 129 2515 1165 1241 333 3002 2655 1990 657 1541 2268 2978 2248
647 451 2081 3052 3135 522 1534 210 3292 1423 751 2627 2738 957 2508 2208 2164 2386 1939 3120 2126 1491 467 523 1201 13
21 1595 165 1636 1511 1430 756 2253 1244 2395 944 552 949 1328 2482 1033 718 2795 678 529 3047 2864 1092 2762 2954 611 3
119 955 2467 2165 2722 907 1162 982 2552 1053 1483 2102 2507 2034 739 2386 518 859 2907 2875 1705 2755 3206 2101 1264 30
0 2443 2374 3260 109 1832 2887 3104 94 1768 137 753 3064 1418 2617 3195 2926 496 2519 2700 2784 950 693 952 995 2196 296
7 2416 2805 414 906 2699 990 2939 885 447 874 2252 2651 2368 3253 2488 1353 2047 110 2833 2654 890 2038 1288 469 765 187
2321 591 1431 963 3032 2505 1372 2556 2948 227 3269 1140 1886 2501 1769 510 1070 245 1571 2176 2924 1089 999 926 3191 1
943 1178 2913 1291 1050 239 785 941 325 3027 3087 2808 1616 1573 3024 354 1200 1289 1460 760 985 3286 24 1584 29 2617 15
3 1199 419 1294 1332 557 659 264 1344 479 277 23 1938 1357 1441 275 3307 2714 2750 2808 2827 1160 111 1758 2673 452 1982
503 3044 1467 2765 3287 2816 1595 2412 361 3136 749 2176 1594 3092 2499 746 2874 1578 1711 2833 1352 333 1879 553
Tous les coefficients sont dans l'anneau modulaire modulo 3329
```

```
Coefficients de l'élément NTT g0n0r0 :
848 1360 3214 1448 522 242 1234 1309 1305 2232 3053 2138 3026 2758 2330 1298 2917 582 1457 1982 3021 1615 3074 3105 2174
2348 3308 1326 1989 1857 2789 1042 2203 1493 3093 1324 1842 859 2010 1291 3055 2364 1686 2911 158 1913 1684 1381 2328 8
10 1567 2796 1928 2700 944 3180 434 2924 998 529 1517 1058 3170 439 3141 153 1469 1691 3119 2393 3188 3229 2337 1633 426
576 653 1813 3045 576 1519 1879 2913 638 1427 591 3133 2558 454 1471 945 2771 3162 1806 1871 2232 581 2988 955 1824 142
7 754 697 2519 1476 1155 1758 1698 1713 1992 3168 81 453 1113 204 996 379 2654 1310 814 2118 1057 2550 1634 1192 1112 26
6 1297 1087 778 1440 2927 335 2994 956 718 502 2133 2663 2831 1792 1625 304 2397 2294 3320 777 1785 1932 3124 1195 2196
1097 3214 667 403 3030 2846 1886 1743 1345 2070 2849 1005 137 2578 2338 3069 530 2093 2518 2574 2400 1791 2035 2861 143
1772 553 3006 1397 668 2712 2757 2102 1653 2637 852 1661 2148 2114 1597 877 2014 1113 2459 374 397 2547 3015 651 2884 20
98 1673 3085 1331 2633 1187 1410 1078 370 862 1448 1173 3315 2538 1343 948 2440 262 2916 2676 3218 3012 1756 280 277 150
6 132 1099 89 1428 1374 3018 2913 1996 1339 2846 1342 2851 2951 2645 2058 2260 2252 2922 673 1629 1993 912 2144 424 2255
62 2145 2504
Tous les coefficients sont dans l'anneau modulaire modulo 3329
```

6. SamplePolyCBD η

La fonction `SamplePolyCBD_eta` de la classe **Poly** est utilisée pour générer un polynôme aléatoire selon la distribution de Bernoulli centrée (CBD) avec le paramètre η .

Le déroulement de la fonction est le suivant :

- La fonction commence par initialiser un objet `BitArray` appelé **b** avec la valeur 0.
- Puis, pour chaque indice **i** allant de 0 à 255 (soit la taille du polynôme **n=256**), la fonction effectue les étapes suivantes :
 - Initialisation de deux variables **x** et **y** à 0.
 - Itération de η afin de générer les bits aléatoires du polynôme. Pour chaque itération **j**, on ajoute le bit à l'indice $2 \times i \times \eta + j$ de **b** à **x**. Ensuite, on ajoute le bit à l'indice $2 \times i \times \eta + \eta + j$ de **b** à **y**. Enfin, nous calculons la valeur du coefficient du polynôme à l'indice **i** comme $(x - y + q) \% q$.
- La fonction stocke cette valeur dans le polynôme **f** à l'indice **i** et retourne le polynôme **f** ainsi généré.

Cette fonction joue un rôle crucial dans la génération de polynômes aléatoires selon une distribution spécifique, ce qui est essentiel pour ce protocole cryptographique.

7. NTT

La fonction spécifiée par le standard a été transposée en tant que constructeur de la classe **NTT**, prenant en entrée un objet **Poly** et calculant la transformée de Fourier rapide (NTT) de ce polynôme. Voici le déroulement de la fonction :

La fonction commence par créer un tableau **f_hat** de taille **n** et le remplit avec les coefficients du polynôme **f**.

Ensuite, la fonction entre dans une boucle **for** qui effectue les calculs de la NTT sur les coefficients. La boucle extérieure parcourt des longueurs de puissance de 2 décroissantes (**len**), de 128 à 2. La boucle intérieure parcourt les indices du tableau **f_hat** par blocs de taille **len**.

À l'intérieur de la boucle, la fonction calcule la NTT. Pour chaque indice **j** dans le bloc courant, la fonction effectue les opérations suivantes :

- Elle calcule la valeur **t** en multipliant la valeur **f_hat[j+len]** par la racine de l'unité **Zeta.getZeta(k)**.
- Elle met à jour **f_hat[j+len]** en soustrayant **t** modulo **q** à **f_hat[j]**.
- Elle met à jour **f_hat[j]** en ajoutant **t** modulo **q** à **f_hat[j]**.

Après la boucle imbriquée, la fonction stocke le résultat final de la NTT dans le tableau coefficients de l'objet NTT.

La fonction utilise des conversions de type (**static_cast<int64_t>**) pour éviter les dépassements de capacité lors des calculs impliquant des valeurs importantes.

Cette implémentation de la NTT est une version optimisée qui utilise la technique Cooley-Tukey pour réduire la complexité de calcul de $O(n^2)$ à $O(n \log n)$. Cela permet une exécution plus efficace, notamment pour les polynômes de grande taille.

```
Input: 41 1822 3005 3197 2524 2408 1491 2726 330 1161 2376 1513 3307 182 3303 491 2995 1955 1498 2107 2430 1288 573 153 292 2395 776
2071 3073 3250 2118 1752 1455 1551 1869 3267 2364 2996 390 3236 2071 508 1361 372 1028 1335 1825 1053 1621 210 2244 1012 2701 2796 63
2872 2065 3083 897 778 2329 3035 2216 1842 288 145 2382 2284 2619 2674 814 502 2574 71 1067 2034 1690 1140 1090 219 2984 2636 781 33
09 2111 1853 1637 718 615 3005 299 2478 1323 1336 2208 1564 2802 2082 2955 3225 1504 1154 1310 3026 2730 3272 661 2306 1712 2412 1692
2113 292 2427 2941 2500 145 2257 1768 3196 683 3303 2503 1991 2381 1464 352 2258 702 2065 718 1150 296 1750 650 101 1146 230 1362 13
50 2141 2300 1121 396 1629 2251 2248 3100 918 1943 3093 1643 874 3043 3097 2536 900 2630 2117 1655 765 3030 992 563 1574 3154 963 712
273 1047 304 875 2716 1033 1267 718 716 3225 3023 1181 1623 1405 53 1999 3115 1306 242 459 1482 467 399 1577 1345 2509 1162 2421 994
3288 2839 2856 993 958 2290 806 626 1920 3203 1595 2824 342 2895 1021 2515 2648 1673 2741 3195 511 3093 1027 562 1587 2682 2845 790
1897 142 3289 606 3151 1482 1965 2944 824 1377 525 498 319 1893 2305 2709 876 1545 3025 1300 2862 3170 2914 2511 1524 2886 2634 969 8
1 354 2672
Output of ntt: 1190 498 2487 1287 3058 1857 547 1878 2804 2056 929 2833 303 1344 2933 2152 1801 2800 1268 2710 1028 1201 731 3281 284
1 3150 2295 955 1108 2957 2972 2743 1732 1934 3048 1544 426 2588 1166 570 2030 1365 985 730 299 778 362 2018 2548 2672 1051 2929 1800
585 439 444 1652 3303 2241 667 3096 2653 1945 1943 1840 729 2977 1802 2775 439 452 412 798 177 291 2653 2656 1911 2226 2096 3287 252
2 3022 2877 658 778 3136 1837 3035 162 2500 2788 2835 1364 2255 2480 74 2044 2178 518 1751 1048 191 311 1574 2652 565 2286 2957 168 9
28 1082 3276 2915 1310 1859 2690 778 878 1041 1683 2296 1089 524 1507 1559 269 2168 1563 1707 3267 1048 504 559 175 1976 1127 151 311
1 3225 2181 1333 2446 2436 3287 82 1055 729 1053 2777 1627 601 1991 50 283 3027 1502 3291 2543 192 2005 2896 918 1105 2949 602 1888 5
32 1460 1045 2436 3115 2500 2818 2617 2795 1237 1917 2063 3324 1555 2438 429 1207 2559 3170 136 2553 1712 1471 1539 3260 2262 1451 23
27 220 1917 18 80 2461 2059 177 2079 2487 2638 731 270 1459 1981 1272 3291 7 2106 855 2560 2486 2091 2747 3292 1505 948 1717 1912 150
9 3304 447 1461 205 3160 3159 2293 3052 346 2166 822 825 1898 3238 2580 3141 2092 1456 122 1829 2418 59 1633 1162 2762 2486 731 1066
2587 89 1773 1627
```

8. NTT-1

La NTT inverse est l'opération inverse de la fonction NTT, permettant de revenir de l'espace des valeurs évaluées vers l'espace des coefficients. Il est inclus dans la classe **Poly** comme constructeur de celle-ci.

Fonctionnement de l'algorithme

L'algorithme de la NTT inverse se déroule en plusieurs étapes :

- Le polynôme d'entrée **f_hat** est copié dans un tableau **f** de **PolyCoef**.
- Une boucle externe parcourt les longueurs de transformée **len** en les divisant par 2 à chaque itération (128, 64, 32, ..., 2).
- Pour chaque longueur de transformée **len**, une boucle interne parcourt les segments de **f** par pas de $2 * len$.
- Pour chaque segment, la valeur de **k** est décrémentée. Cette valeur de **k** est utilisée pour récupérer la valeur de **zeta** correspondante dans le tableau **zetaList**.
- Pour chaque indice **j** dans le segment, les calculs suivants sont effectués :
 - La valeur **t** est initialisée avec la valeur de **f[j]**.
 - La valeur **tmp1** est calculée comme la somme de **f[j]** et **f[j + len]**, modulo **q**.
 - La valeur **f[j]** est mise à jour avec $(tmp1 + q) \% q$.
 - La valeur **tmp2** est calculée comme le produit de **zeta** et la différence entre **f[j + len]** et **t**, modulo **q**.
 - La valeur **f[j + len]** est mise à jour avec $(tmp2 + q) \% q$.
- Après les boucles, chaque coefficient de **f** est multiplié par **3303**, modulo **q**, pour terminer l'algorithme de la NTT inverse.
- Enfin, le tableau **f** contenant les coefficients récupérés est stocké dans l'attribut **coefficients** de la classe **Poly**.

Paramètres et variables

f_hat : le polynôme d'entrée, de type **NTT**, représentant le polynôme dans l'espace des valeurs évaluées.

k : un indice utilisé pour récupérer les valeurs de zeta dans le tableau **zetaList**.

len : la longueur de transformée courante, variant de 2 à 128.

start : l'indice de début du segment courant dans le tableau **f**.

j : l'indice courant dans le segment.

t, **tmp1** et **tmp2** : des variables intermédiaires utilisées dans les calculs.

zetaList : un tableau contenant les racines n-ièmes de l'unité modulo **q**.

La complexité de l'algorithme de la NTT inverse est en $O(n \log n)$, où **n** est la taille du polynôme. Cela en fait une opération relativement efficace, permettant de traiter des polynômes de grande taille.

9. BitRev7

La fonction BitRev effectue une opération appelée "inversion de bits" initialement sur un nombre entier non signé de 7 bits. Toutefois, elle a été adaptée pour fonctionner indépendamment de la taille de l'entrée. L'inversion de bits consiste à réorganiser l'ordre des bits dans la représentation binaire d'un nombre. Concrètement, BitRev prend un nombre entier de xx bits en entrée et retourne un nouveau nombre où l'ordre des bits a été inversé. Par exemple, si l'entrée est le nombre binaire 0111000 (56 en décimal), la sortie sera 0001110 (14 en décimal). Un autre exemple, qui ne prend pas en compte la notation binaire serait 1234567 en entrée, qui devient 7654321 en sortie. Cette opération est utile dans certains algorithmes de traitement de signal numérique.

La méthode BitRev de la classe **ZETA** inverse le sens de lecture d'un octet selon un nombre donné de bits. Elle prend en entrée l'**octet** à inverser et **numBits**, le nombre de bits à considérer pour l'inversion puis procède comme suit :

- Parcourt les bits de l'**octet** de droite et gauche en même temps.
- Dans la variable **reversedOctet**, les bits de droite sont enregistrés à gauche symétriquement par rapport au milieu de l'octet. Inversement pour les bits de gauche.

Le résultat retourné est l'octet inversé.

10. findNumBits

La méthode findNumBits de la classe **ZETA** calcule le nombre de bits nécessaires pour représenter la moitié de **n**. Si **n** = 256 alors 7 bits sont nécessaires. Cette méthode décale les bits de **nTmp** vers la droite jusqu'à trouver le nombre de bits nécessaire pour représenter la moitié de **n**. Cette méthode ne fait pas partie du document du FIPS, elle a été ajoutée pour adapter M-LKEM à d'autres constantes **n**, **q** et **ζ**. La variable retournée est **numBits**, qui s'incrémente à chaque décalage de bits.

11. findSmallestPrimitiveRoot

La méthode findSmallestPrimitiveRoot de la classe **ZETA** recherche la plus petite racine primitive de l'unité d'ordre **n** modulo **q**, noté **ζ** dans le document du FIPS. Cette méthode ne fait pas partie du document du FIPS, elle a été ajoutée pour adapter M-LKEM à d'autres constantes **n**, **q** et **ζ**. Le premier entier **i** qui vérifie $i^{\frac{n}{2}} \equiv q - 1 \pmod{q}$ est retournée.

12. Zeta

Cette méthode est le constructeur de la classe Zeta, elle calcule à la suite toutes les valeurs de $\zeta^{\text{BitRev}(k)}$ et $\zeta^{2 \cdot \text{BitRev}(k)+1}$ pour $k \in \{0, 1, \dots, \frac{n}{2} - 1\}$ et **ζ** une racine primitive de l'unité d'ordre **n** modulo **q**. Toutes les valeurs sont stockées dans deux listes zetaList et gammaList accessibles depuis le fichier d'en-tête constants.hpp

13. Surcharge d'opérateur * NTT

La surcharge d'opérateur * de la classe **NTT** est utilisée pour multiplier deux objets **NTT** ensemble. Elle rassemble les fonction `MultiplyNTTs` et `BaseCaseMultiply` du document du FIPS.

Paramètres et variables :

f, g : Les deux objets **NTT** à multiplier ensemble.

gamma : Une puissance d'une racine primitive de l'unité d'ordre n modulo q.

Fonctionnement :

- L'objet **NTT h** est initialisé pour stocker le résultat de la multiplication.
- A chaque itération de la boucle, les coefficients des polynômes **f** et **g** aux indices pairs et impairs sont extraits dans **a0, a1, b0, b1**.
- Calcule le produit de deux polynômes de degré 1 modulo un polynôme quadratique de la forme $X^2 - \text{gamma}$ et modulo un entier q. En effectuant une multiplication polynomiale de base entre les coefficients **a0, a1, b0, b1** et **gamma**.

$$c0 = a0 \cdot b0 + a1 \cdot b1 \cdot \gamma$$

$$c1 = a0 \cdot b1 + a1 \cdot b0$$

- Les coefficients résultants **c0** et **c1** sont ensuite mis à jour dans l'objet **h**.

D. K-PKE

Cette section concerne les algorithmes de chiffrement et de génération de clés qui composent le ML-KEM. C'est là que toutes les briques et méthodes détaillées précédemment s'imbriquent pour former les algorithmes. Grâce à certaines surcharges, l'implémentation peut suivre au plus près la syntaxe du standard tel qu'il est écrit aujourd'hui. Cependant, des incohérences et des inconsistances dans la documentation ont ralenti voire bloqué le développement de ces parties.

14. K-PKE.KeyGen

A partir de cet élément, l'implémentation s'est arrêtée. Cependant, grâce à une bonne base établie précédemment, l'assemblage semble simplifié. Il reste cependant des cas qui n'ont pas été anticipés.

- Ligne 17 et 18 : il faut ajouter une méthode à **NTTMatrice** pour passer d'une **PolyMatrice** à une **NTTMatrice**, avec pourquoi pas un constructeur.
- Ligne 19 : Cette ligne qui peut sembler simple à première vue, est plus complexe d'un point de vu algorithmique. Cependant, grâce aux surcharges effectuées, son écriture sera proche de cette syntaxe.
- Ligne 20 : Il faut implémenter une méthode de **ByteEncode** qui prend en entrée une **NTTMatrice** et renvoie un **ByteArray** qui contient toutes les **NTT** encodés puis concaténées à la suite l'une de l'autre. Puis implémenter une surcharge pour concaténer un **ByteArray** avec **p**

15. K-PKE.Encrypt

Bien que cette partie n'ait pas été évaluée, elle a soulevé des interrogations concernant notre structure. En effet, la ligne 21 effectue une multiplication entre deux matrices NTT, puis convertit le résultat en une matrice de polynômes, ce qui ne devrait pas poser de problème jusqu'à présent. Finalement, une somme de matrices de polynômes et d'un polynôme simple est réalisée, ce qui suscite des questions sur la manière de procéder, qui n'est décrite nulle part.

E.Améliorations

Pour garantir la maintenabilité de ce travail, plusieurs aspects doivent être pris en compte. Tout d'abord, l'ajout de commentaires serait bénéfique pour faciliter l'utilisation du code, notamment au niveau des méthodes des classes. Ces commentaires devraient expliquer leur fonctionnement ainsi que leur correspondance avec les éléments du standard, en fournissant par exemple des références numériques.

Ensuite, il est nécessaire d'améliorer les tests unitaires en incluant davantage de jeux de données. Cela permettra de s'assurer que les fonctions fonctionnent comme prévu et retournent des valeurs correctes.

Il est également important d'ajouter des tests d'intégration pour vérifier la concordance des fonctions, en s'assurant que les types de variables d'entrée et de sortie correspondent sans provoquer d'erreurs de compatibilité. Cela devrait être fait pour toutes les méthodes de chaque classe.

Il convient de revoir les méthodes de chaque classe pour évaluer leur utilité et en implémenter de nouvelles si nécessaire, au fur et à mesure que de nouveaux besoins se font sentir. Par exemple, la transposée de matrice est utilisée à plusieurs reprises dans certains algorithmes K-PKE.

Une amélioration du rapport lui-même est également nécessaire. Certaines explications se contentent de décrire ce qui est déjà visible dans le code, sans mettre suffisamment l'accent sur les aspects techniques ou calculatoires de la fonction. De plus, les choix d'implémentation devraient être expliqués en détail, en mettant en lumière le besoin de certaines surcharges d'opérateurs, le type de données en jeu et leur utilité dans le développement.

Enfin, dans une optique d'implémentation didactique, il aurait été judicieux d'expliquer les principes mathématiques étape par étape. Par exemple, en prenant un petit polynôme et en calculant sa convolution de manière classique, puis en le réalisant dans le domaine de la NTT pour comprendre visuellement le fonctionnement et l'intérêt de cette utilisation.

Il semble que l'objectif principal a parfois été négligé, concentrant plutôt les efforts sur le développement technique. Une approche plus équilibrée entre le développement et l'explication des principes sous-jacents aurait été bénéfique.

F. Conclusion

Ce rapport technique offre une plongée approfondie dans l'implémentation de l'algorithme ML-KEM, un nouveau standard cryptographique conçu pour résister à la menace croissante des ordinateurs quantiques. À travers une exploration rigoureuse des choix techniques, des structures de données et des algorithmes auxiliaires, cette documentation fournit une base solide pour l'implémentation pratique de ML-KEM.

Le choix du langage C++ s'est révélé essentiel pour atteindre les performances et la modularité requises, malgré les défis associés à sa complexité. En utilisant une approche orientée objet, chaque entité et opération a été soigneusement encapsulée dans des classes, assurant ainsi la clarté du code, sa maintenabilité et sa robustesse.

Les dépendances et les fonctions auxiliaires ont été soigneusement intégrées pour compléter l'implémentation principale, bien que des améliorations puissent encore être apportées pour renforcer la documentation et les tests unitaires. L'accent a été mis sur la clarté du code et la concordance avec les spécifications du NIST, bien que certaines parties du développement aient soulevé des questions nécessitant une réflexion supplémentaire.

En conclusion, ce rapport marque une étape importante dans la mise en œuvre pratique de ML-KEM, offrant une base technique solide pour de prochaines implémentations. Les recommandations pour l'amélioration future visent à renforcer la qualité et la compréhension du code, ainsi qu'à approfondir la documentation pour faciliter son utilisation et sa maintenance à long terme.

G. Références

NIST, “*FIPS 203 (Draft)*”, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf>, 20/09/23

GitHub, ML-KEM, <https://github.com/Benoit62/MK-KEM>, 21/04/2024