Implémentation Pythonique d'un Chiffrement Post-Quantique : ML-KEM comme Mécanisme d'Encapsulation de Clés

Épreuve de TIPE

Charaf Ddine El Omari

Session 2025

Sommaire

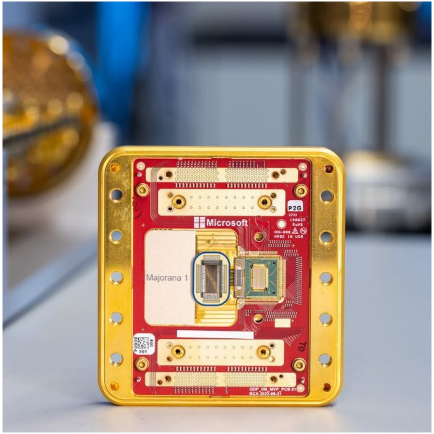


Figure 1 : Majorana 1, 2025

Introduction

Définition

- Cryptographie : Science utilisant les mathématiques pour sécuriser l'information.

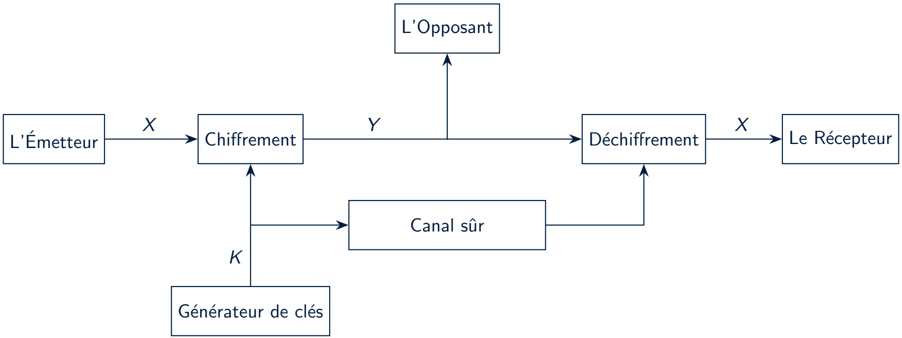


Figure 2 : Le canal de communication

Chiffrements symétriques et asymétriques

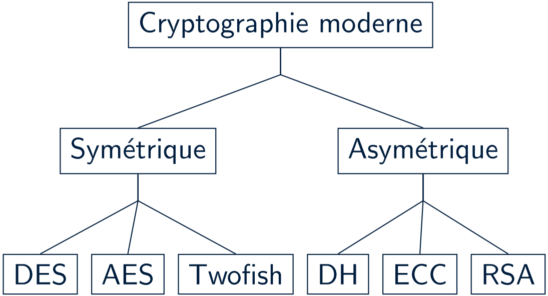


Figure 3 : Exemples de chiffrements modernes

- **Symétrique** : Une seule clé K.
- **Asymétrique** :
  - Une paire de clés (P_k, S_k).
  - Repose sur des problèmes mathématiques.

Chronologie

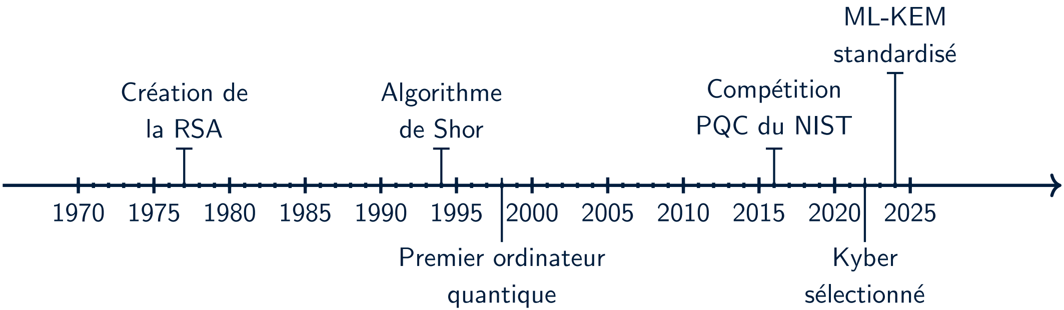


Figure 4 : Évolution vers ML-KEM

Problématique

Comment implémenter ML-KEM correctement ?

Préliminaire mathématique

Catégories de sécurité

	q	n	k	ζ	η_1	η_2	d_u	d_v
ML-KEM-512	3329	256	2	17	3	2	10	4
ML-KEM-768	3329	256	3	17	2	2	10	4
ML-KEM-1024	3329	256	4	17	2	2	11	5

Table 1 : Ensembles de paramètres approuvés pour ML-KEM

Notations

- $\mathbb{Z}_q$: L'anneau $\mathbb{Z}/q\mathbb{Z}$
- $R_q = \mathbb{Z}_q[X]/(X^n + 1)$
- $S_\eta = \{P \in R_q \mid \forall\, i,\ P_i \in [\![-\eta, \eta]\!]\}$
- E^i, $E^{i \times i}$
- **E** : L'Émetteur
- **R** : Le Récepteur

N.B.

Pour fixer les idées, nous travaillerons désormais avec les paramètres de ML-KEM-768.

L'arrondissement dans $\mathbb{Z}_q$

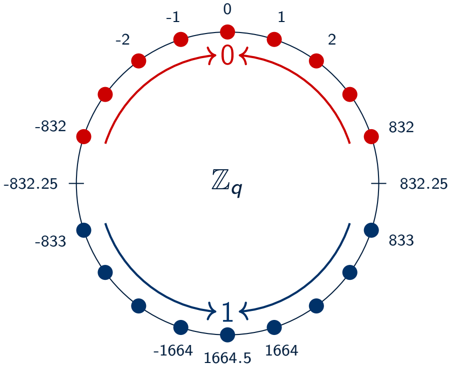


Figure 5 : Visualisation de l'arrondissement dans $\mathbb{Z}_q$

- Soit $x \in [\![0, q-1]\!]$;
- Soit $x' = x \ mod \ q$ tel que $x' \in [\![-\frac{q-1}{2}, \frac{q-1}{2}]\!]$;
- $\text{Arrondi}_q(x) = \begin{cases} 0, & \text{si } |x'| < \frac{q}{4}, \\ 1, & \text{sinon.} \end{cases}$

K-PKE élémentaire

Génération de clés

Ce que fait R :

1. Sélectionner $A \in_u R_q^{k\times k}$, $s \in_u S_{\eta_1}^k$, et $e \in_u S_{\eta_2}^k$.
2. Calculer $t = As + e$.
3. La clé publique de **R** est (A, t) ; sa clé privée est s.

Remarque

Calculer s à partir de (A, t) est un cas de M-LWE.

Chiffrement et Déchiffrement

Pour chiffrer un message $m \in \{0,1\}^n$ pour R, E fait :

1. Obtenir une copie authentique de la clé de chiffrement (A, t) de **R**.
2. Sélectionner $y \in_u S^k_{\eta_1}$, $e_1 \in_u S^k_{\eta_2}$ et $e_2 \in_u S_{\eta_2}$.
3. Calculer $u = A^T y + e_1$ et $v = t^T y + e_2 + \lceil \frac{q}{2} \rfloor m$.
4. Sortir $c = (u, v)$.

Pour déchiffrer $c = (u, v)$, R fait :

Calculer $m = Arrondi_q(v - s^T u)$.

Optimisations

Taille de la clé de chiffrement

- La taille de (A, t) est 4,608 octets.

Démarche

1. Sélectionner une graine aléatoire $\rho \in_u \{0, 1\}^{256}$.
2. Générer les coefficients des polynômes de A en hachant ρ avec un compteur.

- La clé de chiffrement est maintenant (ρ, t) au lieu de (A, t).
- La taille de (ρ, t) est 1,184 octets.

Taille du texte chiffré

- La taille de $c = (u, v)$ est 1,536 octets.

La compression

Soit $1 \leq d < \lceil \log_2 q \rceil$;

- $\forall\, x \in [\![0, q-1]\!],\ Compress_q(x, d) = \lceil \frac{2^d}{q} \cdot x \rfloor \ mod\ 2^d$.
- $\forall\, y \in [\![0, 2^d - 1]\!],\ Décompress_q(y, d) = \lceil \frac{q}{2^d} \cdot y \rfloor \ mod\ q$.

- u et v sont remplacés par $c_1 = Compress_q(u, d_u)$ et $c_2 = Compress_q(v, d_v)$.
- la taille du texte chiffré compressé est 1,088 octets.

Loi binomiale centrée

- Tirer uniformément dans $[\![-\eta,\eta]\!]$ requiert une forme d'échantillonnage par rejet.

Idée

Pour simplifier, les coefficients sont plutôt tirés suivant la loi binomiale centrée :

$$\begin{cases} X(\Omega) = [\![-\eta,\eta]\!] \\ \forall j \in [\![-\eta,\eta]\!];\ P(X=j) = \binom{2\eta}{\eta+j}\frac{1}{2^{2\eta}} \end{cases}$$

La transformation théorique de nombre

Définition

La NTT est un isomorphisme définie comme suit :

$$\begin{aligned} NTT : R_q &\to T_q = \mathcal{Q}_0 \times \mathcal{Q}_1 \times \ \dots \ \times \mathcal{Q}_{127} \\ P &\mapsto \hat{P} = (P \ mod\,(X^2 - \zeta_0), \dots, P \ mod\,(X^2 - \zeta_{127})) \end{aligned}$$

avec $\forall\, i \in [\![0, 127]\!]$; $\mathcal{Q}_i = \mathbb{Z}_q[X]/(X^2 - \zeta_i)$ et $\zeta_i = \zeta^{2 \times IB_7(i)+1}$.

Remarque

La complexité de la NTT est $O(n \log(n))$.

Multiplication rapide de polynômes

- Le temps de chiffrement et de déchiffrement est dominé par la multiplication des polynômes.
- La complexité de l'algorithme classique de multiplication dans R_q est $O(n^2)$.

Multiplication avec la NTT

Pour calculer $C = P \cdot Q \; mod \, (X^n + 1)$ avec $P,\, Q \in R_q$:

1. On Calcul $\hat{P} = NTT(P)$ et $\hat{Q} = NTT(Q)$.
2. On Calcul $\hat{C} = \hat{P} \times_{T_q} \hat{Q}$.
3. On Calcul $C = NTT^{-1}(\hat{C})$.

- La complexité totale est $O(n \log(n))$.

K-PKE complet

Génération de clés

Ce que fait R :

1. Sélectionner $\rho \in_u \{0,1\}^{256}$ et calculer $A = \acute{E}tendre(\rho) \in R_q^{k \times k}$.
2. Sélectionner $s \in_{DBC} S_{\eta_1}^k$, et $e \in_{DBC} S_{\eta_2}^k$.
3. Calculer $t = As + e$.
4. La clé publique de **R** est (ρ, t) ; sa clé privée est s.

k-pke-génclés(d)

```
def k_pke_génclés(d):
    rho, sigma = G(d + bytes([k]))
    N = 0
    A_ch = [[] for i in range(k)]
    for i in range(k):
        for j in range(k):
            A_ch[i].append(ech_ntt(rho + bytes([j, i])))
    s = []
    for i in range(k):
        s.append(ech_dbc(eta1, fpa(eta1, sigma, N)))
        N += 1
    e = []
    for i in range(k):
        e.append(ech_dbc(eta1, fpa(eta1, sigma, N)))
        N += 1
    s_ch = [ntt(poly) for poly in s]
    e_ch = [ntt(poly) for poly in e]
    t_ch = e_ch
    for i in range(k):
        for j in range(k):
            t_ch[i] = ajt_poly(t_ch[i], mul_ntts(A_ch[i][j],s_ch[j]))
    ek_pke = enc_octs(12, t_ch) + rho
    dk_pke = enc_octs(12, s_ch)
    return ek_pke, dk_pke
```

Chiffrement

Pour chiffrer un message $m \in \{0,1\}^n$ pour R, E fait :

1. Obtenir une copie authentique de la clé de chiffrement (ρ, t) de **R**, puis calculer $A = \acute{E}tendre(\rho)$.
2. Sélectionner $y \in_{DBC} S^k_{\eta_1}$, $e_1 \in_{DBC} S^k_{\eta_2}$ et $e_2 \in_{DBC} S_{\eta_2}$.
3. Calculer $u = A^T y + e_1$ et $v = t^T y + e_2 + \lceil \frac{q}{2} \rfloor m$.
4. Calculer $c_1 = Compress_q(u, d_u)$ et $c_2 = Compress_q(v, d_v)$.
5. Sortir $c = (c_1, c_2)$.

k-pke-chiffrer(ek-pke, m, r)

```
def k_pke_chiffrer(ek_pke, m, r):
    N = 0
    t_ch = [dec_octs(12, ek_pke[384*i:384*(i+1)]) for i in range(k)]
    rho = ek_pke[384*k : 384*k + 32]
    A_ch = [[] for i in range(k)]
    for i in range(k):
        for j in range(k):
            A_ch[i].append(ech_ntt(rho + bytes([j, i])))
    y = []
    for i in range(k):
        y.append(ech_dbc(eta1, fpa(eta1, r, N)))
        N += 1
    e1 = []
    for i in range(k):
        e1.append(ech_dbc(eta2, fpa(eta2, r, N)))
        N += 1
    e2 = ech_dbc(eta2, fpa(eta2, r, N))
    y_ch = [ntt(poly) for poly in y]
```

k-pke-chiffrer(ek-pke, m, r)

```
        u = [[0]*256 for i in range(k)]
        for i in range(k):
            for j in range(k):
                u[i] = ajt_poly(u[i], mul_ntts(A_ch[j][i], y_ch[j]))
        for i in range(k):
            u[i] = inv_ntt(u[i])
            u[i] = ajt_poly(u[i], e1[i])
        mu = décomp(1, dec_octs(1, m))
        v = [0]*256
        for i in range(k):
            v = ajt_poly(v, mul_ntts(t_ch[i], y_ch[i]))
        v = inv_ntt(v)
        v = ajt_poly(v, e2)
        v = ajt_poly(v, mu)
        c1 = b''
        for i in range(k):
            c1 += enc_octs(du, comp(du, u[i]))
        c2 = enc_octs(dv, comp(dv, v))
        c = c1 + c2
        return c
```

Déchiffrement

Pour déchiffrer $c = (c_1, c_2)$, R fait :

1. Calculer $u' = Décompress_q(c_1, d_u)$ et $v' = Décompress_q(c_2, d_v)$.
2. Calculer $m = Arrondi_q(v' - s^T u')$.

Remarque

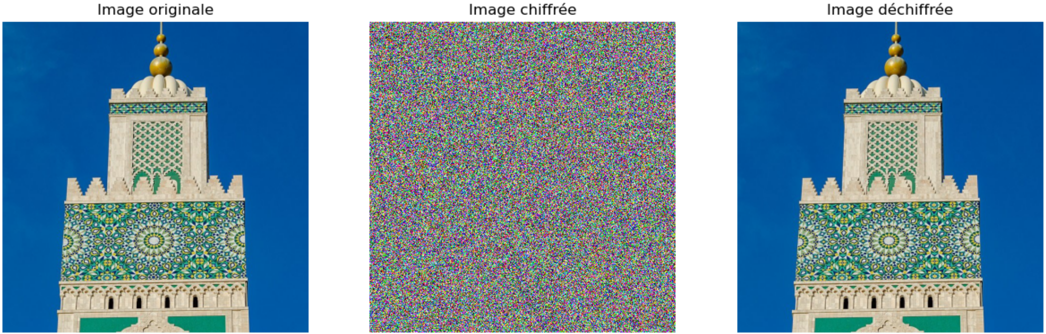
Kyber-PKE n'est pas destiné à une utilisation autonome.

k-pke-déchiffrer(dk-pke, c)

```
def k_pke_déchiffrer(dk_pke, c):
    c1 = c[0 : 32*du*k]
    c2 = c[32*du*k : 32*(du*k + dv)]
    up = [décomp(du, dec_octs(du, c1[32*du*i : 32*du*(i+1)])) for i in range(k)]
    vp = décomp(dv, dec_octs(dv, c2))
    s_ch = [dec_octs(12, dk_pke[384*i:384*(i+1)]) for i in range(k)]
    w = [0]*256
    for i in range(k):
        w = ajt_poly(w, mul_ntts(s_ch[i], ntt(up[i])))
    w = sous_poly(vp, inv_ntt(w))
    m = enc_octs(1, comp(1, w))
    return m
```

Test visuel

- Problème : décalage de pixels. Solution : padding.
- Affichage des résultats via Matplotlib :



- Hypothèse : L'algorithme K-PKE est implémenté correctement.

ML-KEM

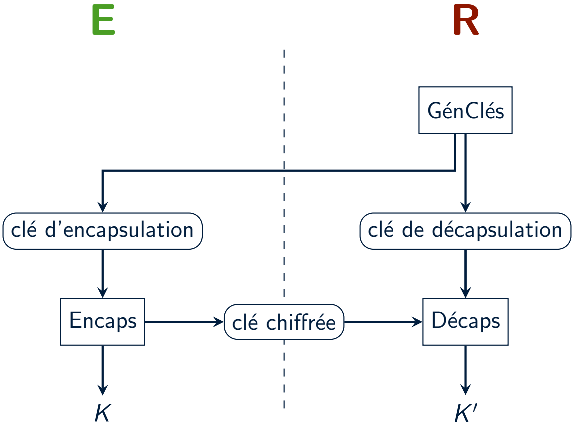


Figure 6 : Établissement de clé à l'aide d'un KEM

La transformation de Fujisaki-Okamoto

Définition

La transformation FO est une méthode générique pour atteindre le niveau de sécurité IND-CCA. Elle met en œuvre trois algorithmes de hachage :

$$G : \{0,1\}^* \to \{0,1\}^{512}$$
$$H : \{0,1\}^* \to \{0,1\}^{256}$$
$$J : \{0,1\}^* \to \{0,1\}^{256}$$

Génération de clés

R fait :

1. Générer les clés de chiffrement et de déchiffrement en utilisant K-PKE.
2. Sélectionner $z \in_u \{0,1\}^{256}$.
3. La clé d'encapsulation de **R** est $ek = (\rho, t)$; sa clé de décapsulation est $dk = (s, ek, H(ek), z)$.

ML-KEM-GénClés()

```
def génclés_interne(d, z):
    (ek_pke, dk_pke) = k_pke_génclés(d)
    ek  = ek_pke
    dk  = dk_pke + ek + H(ek) + z
    return (ek, dk)
```

```
def ML_KEM_GénClés():
    d = os.urandom(32)
    z = os.urandom(32)
    if d == None or z == None:
        return "Échec de la génération de bits aléatoires"
    (ek, dk) = génclés_interne(d, z)
    return (ek, dk)
```

Encapsulation

Pour établir une clé secrète partagée avec R, E fait :

1. Obtenir une copie authentique de la clé d'encapsulation de **R**, ek.
2. Sélectionner $m \in_u \{0,1\}^{256}$.
3. Calculer $h = H(ek)$ et $(K, r) = G(m, h)$, avec $K, r \in \{0,1\}^{256}$.
4. Calculer $c = k\_pke\_chiffrer(ek, m, r)$.
5. Sortir la clé secrète K et le texte chiffré c.

ML-KEM-Encaps(ek)

```python
def encaps_interne(ek, m):
    (k, r)  = G(m + H(ek))
    c = k_pke_chiffrer(ek, m, r)
    return  (k, c)
```

```python
def ML_KEM_Encaps(ek):
    m = os.urandom(32)
    if m == None:
        return "Échec de la génération de bits aléatoires"
    (k, c) = encaps_interne(ek, m)
    return  (k, c)
```

Décapsulation

Pour récupérer la clé secrète K à partir de c en utilisant dk, R fait :

1. Calculer $m' = k\_pke\_déchiffrer(s, c)$.
2. Calculer $(K', r') = G(m', H(ek))$.
3. Calculer $\bar{K} = J(z, c)$.
4. Calculer $c' = k\_pke\_chiffrer(ek, m', r')$.
5. Si $c \neq c'$, retourner $\bar{K}$, sinon retourner K'.

ML-KEM-Décaps(dk, c)

```
def décaps_interne(dk, c):
    dk_pke = dk[0 : 384*k]
    ek_pke = dk[384*k : 768*k + 32]
    h  = dk[768*k + 32 : 768*k + 64]
    z  = dk[768*k + 64 : 768*k + 96]
    mp = k_pke_déchiffrer(dk_pke, c)
    (kp, rp) = G(mp + h)
    kb = J(z + c)
    cp = k_pke_chiffrer(ek_pke, mp, rp)
    if c != cp:
        kp = kb
    return kp
```

```
def ML_KEM_Décaps(dk, c):
    kp = décaps_interne(dk, c)
    return kp
```

Test final

Tests à réponses connues

- On obtient les vecteurs de test depuis le dépôt officiel du NIST.
- On compare les sorties avec les résultats attendus :

```
sharaf@debian:~/MLKEM$ python3 MLKEM.py
 ML-KEM GénClés : RÉUSSI= 75  ÉCHOUÉ= 0
 ML-KEM Encaps  : RÉUSSI= 75  ÉCHOUÉ= 0
 ML-KEM Décaps  : RÉUSSI= 30  ÉCHOUÉ= 0
 ML-KEM  -- Total ÉCHOUÉ= 0
```

- Tous les résultats concordent : l'implémentation est validée.

Conclusion

- Pour atteindre sa forme finale, K-PKE doit subir de nombreuses optimisations.
- ML-KEM est ensuite obtenu en appliquant la transformation de Fujisaki-Okamoto.
- Les tests à réponses connues nous ont permis de confirmer la justesse de l'ensemble de l'implémentation. L'hypothèse de l'implémentation correcte de K-PKE est donc vérifiée.
- Prochaine étape : sécurisation contre les attaques par canaux auxiliaires.

Merci Pour Votre Écoute

Annexe A — MLKEM.py

Importations, paramètres, et précalculs pour la NTT

```
from test_mlkem import test_mlkem # Obtenu à partir de [6]
from Crypto.Hash import SHAKE128, SHAKE256, SHA3_256, SHA3_512
import os

ML_KEM_PARAM = {
    "ML-KEM-512"  : ( 2, 3, 2, 10, 4 ),
    "ML-KEM-768"  : ( 3, 2, 2, 10, 4 ),
    "ML-KEM-1024" : ( 4, 2, 2, 11, 5 )
}

ML_KEM_ZETA_NTT = [
    1,       1729,   2580,   3289,   2642,   630,    1897,   848,
    1062,    1919,   193,    797,    2786,   3260,   569,    1746,
    296,     2447,   1339,   1476,   3046,   56,     2240,   1333,
    1426,    2094,   535,    2882,   2393,   2879,   1974,   821,
    289,     331,    3253,   1756,   1197,   2304,   2277,   2055,
    650,     1977,   2513,   632,    2865,   33,     1320,   1915,
    2319,    1435,   807,    452,    1438,   2868,   1534,   2402,
    2647,    2617,   1481,   648,    2474,   3110,   1227,   910,
    17,      2761,   583,    2649,   1637,   723,    2288,   1100,
    1409,    2662,   3281,   233,    756,    2156,   3015,   3050,
    1703,    1651,   2789,   1789,   1847,   952,    1461,   2687,
    939,     2308,   2437,   2388,   733,    2337,   268,    641,
    1584,    2298,   2037,   3220,   375,    2549,   2090,   1645,
    1063,    319,    2773,   757,    2099,   561,    2466,   2594,
    2804,    1092,   403,    1026,   1143,   2150,   2775,   886,
    1722,    1212,   1874,   1029,   2110,   2935,   885,    2154 ]
```

Précalculs pour la multiplication NTT

```
ML_KEM_ZETA_MUL = [
    17,     -17,    2761,   -2761,  583,    -583,   2649,   -2649,
    1637,   -1637,  723,    -723,   2288,   -2288,  1100,   -1100,
    1409,   -1409,  2662,   -2662,  3281,   -3281,  233,    -233,
    756,    -756,   2156,   -2156,  3015,   -3015,  3050,   -3050,
    1703,   -1703,  1651,   -1651,  2789,   -2789,  1789,   -1789,
    1847,   -1847,  952,    -952,   1461,   -1461,  2687,   -2687,
    939,    -939,   2308,   -2308,  2437,   -2437,  2388,   -2388,
    733,    -733,   2337,   -2337,  268,    -268,   641,    -641,
    1584,   -1584,  2298,   -2298,  2037,   -2037,  3220,   -3220,
    375,    -375,   2549,   -2549,  2090,   -2090,  1645,   -1645,
    1063,   -1063,  319,    -319,   2773,   -2773,  757,    -757,
    2099,   -2099,  561,    -561,   2466,   -2466,  2594,   -2594,
    2804,   -2804,  1092,   -1092,  403,    -403,   1026,   -1026,
    1143,   -1143,  2150,   -2150,  2775,   -2775,  886,    -886,
    1722,   -1722,  1212,   -1212,  1874,   -1874,  1029,   -1029,
    2110,   -2110,  2935,   -2935,  885,    -885,   2154,   -2154 ]
```

Initialisation de la classe, fonctions de hachage, et fonction pseudo-aléatoire

```python
class ML_KEM:

    def __init__(self, param='ML-KEM-768'):
        if param not in ML_KEM_PARAM:
            raise ValueError
        self.q = 3329
        self.n = 256
        (self.k, self.eta1, self.eta2, self.du, self.dv) = ML_KEM_PARAM[param]

    def H(self, x): return SHA3_256.new(x).digest()

    def G(self, x): h = SHA3_512.new(x).digest(); return (h[0:32], h[32:64])

    def J(self, s): return SHAKE256.new(s).read(32)

    def fpa(self, eta, s, b): return SHAKE256.new(s + bytes([b])).read(64*eta)
```

Compression, décompression et conversions bits-octets

```
    def comp(self, d, xv):
        return [((x * (2**d) + (self.q-1)//2) // self.q) % (2**d) for x in xv]

    def décomp(self, d, yv):
        return [(self.q*y + (1 * (2**(d - 1)))) // (2**d) for y in yv]

    def bits_vers_octs(self, b):
        B = [0] * (len(b) // 8)
        for i in range(len(b)):
            B[i // 8] += b[i] * (2 ** (i % 8))
        return bytearray(B)

    def octs_vers_bits(self, B):
        b = [0] * (len(B) * 8)
        C = bytearray(B)
        for i in range(len(C)):
            for j in range(8):
                b[8 * i + j] = C[i] % 2
                C[i] //= 2
        return b
```

Encodage et décodage

```python
    def enc_octs(self, d, F):
        if type(F[0]) == list: return b''.join(self.enc_octs(d, x) for x in F)
        m = 2 ** d if d < 12 else self.q
        b = [0] * (256 * d)
        for i in range(256):
            a = F[i]
            for j in range(d):
                b[i * d + j] = a % 2
                a = (a - b[i * d + j]) // 2
        return self.bits_vers_octs(b)

    def dec_octs(self, d, B):
        m = 2**d if d < 12 else self.q
        b = self.octs_vers_bits(bytearray(B))
        F = [0] * 256
        for i in range(256):
            for j in range(d):
                F[i] += b[i * d + j] * (2 ** j) %m
        return F
```

Algorithmes d’échantillonnage

```
    def ech_ntt(self, B):
        xof = SHAKE128.new(B)
        j, a_ch = 0, []
        while j < 256:
            C = xof.read(3)
            d1 = C[0] + 256*(C[1] % 16)
            d2 = (C[1] // 16) + 16*C[2]
            if d1 < self.q:
                a_ch += [ d1 ]
                j += 1
            if d2 < self.q and j < 256:
                a_ch += [ d2 ]
                j += 1
        return a_ch

    def ech_dbc(self, eta, B):
        b = self.octs_vers_bits(B)
        f = [0]*256
        for i in range(256):
            x = sum(b[2*i*eta:(2*i + 1)*eta])
            y = sum(b[(2*i + 1)*eta:2*(i + 1)*eta])
            f[i] = (x - y) % self.q
        return f
```

La NTT (Number-Theoretic Transform)

```
    def ntt(self, f):
        f_ch = f.copy()
        i, len = 1, 128
        while len >= 2:
            for st in range(0, 256, 2*len):
                ze = ML_KEM_ZETA_NTT[i]; i += 1
                for j in range(st, st+len):
                    t = (ze*f_ch[j + len]) % self.q
                    f_ch[j + len] = (f_ch[j] - t) % self.q
                    f_ch[j] = (f_ch[j] + t) % self.q
            len //= 2
        return f_ch

    def inv_ntt(self, f_ch):
        f = f_ch.copy()
        i, len = 127, 2
        while len <= 128:
            for st in range(0, 256, 2*len):
                ze = ML_KEM_ZETA_NTT[i]; i -= 1
                for j in range(st, st+len):
                    t = f[j]
                    f[j] = (t + f[j + len]) % self.q
                    f[j + len] = (ze*(f[j + len] - t)) % self.q
            len *= 2
        return [(x*3303) % self.q for x in f]
```

Multiplication NTT et opérations sur les polynômes

```python
    def mul_ntts(self, f_ch, g_ch):
        h_ch = []
        for i in range(0, 128):
            h_ch += self.cas_base_mul(f_ch[2*i], f_ch[2*i+1], g_ch[2*i], g_ch[2*i+1], ML_KEM_ZETA_MUL[i])
        return h_ch

    def cas_base_mul(self, a0, a1, b0, b1, gam):
        return (a0*b0 + a1*b1*gam) % self.q, (a0*b1 + a1*b0) % self.q

    def ajt_poly(self, f, g): return [(f[i] + g[i]) % self.q for i in range(256)]

    def sous_poly(self, f, g): return [(f[i] - g[i]) % self.q for i in range(256)]
```

Le test final

```python
    #def k_pke_génclés(self, d): ...
    #def k_pke_chiffrer(self, ek_pke, m, r): ...
    #def k_pke_déchiffrer(self, dk_pke, c): ...

    #def génclés_interne(self, d, z, param=None):
    #    if param: self.__init__(param) ...
    #def encaps_interne(self, ek, m, param=None):
    #    if param: self.__init__(param) ...
    #def décaps_interne(self, dk, c, param=None):
    #    if param: self.__init__(param) ...

    #def ML_KEM_GénClés(self): ...
    #def ML_KEM_Encaps(self, ek): ...
    #def ML_KEM_Décaps(self, dk, c): ...

if __name__ == '__main__':
    mk = ML_KEM()
    test_mlkem( mk.génclés_interne,
                mk.encaps_interne,
                mk.décaps_interne,
                )
```

Annexe B — Test-image.py

Importations et génération de clés

```
from MLKEM import ML_KEM
import matplotlib.pyplot as plt
import numpy as np
import random as rd
import os


mk = ML_KEM(param='ML-KEM-1024') # Nous avons choisi ML-KEM-1024 afin de conserver les proportions de l'image.

d = os.urandom(32)
ek, dk = mk.k_pke_génclés(d)
```

chiffrer-img(A)

```
def chiffrer_img(A):
    l,c,_=A.shape
    chiffré = []
    R=[]
    G=[]
    B=[]
    for i in range(l):
        for j in range(c):
            if len(R)<32:
                R.append(A[i][j][0])
                G.append(A[i][j][1])
                B.append(A[i][j][2])
                continue
            Rc=bytearray(mk.k_pke_chiffrer(ek, R, os.urandom(32)))
            Gc=bytearray(mk.k_pke_chiffrer(ek, G, os.urandom(32)))
            Bc=bytearray(mk.k_pke_chiffrer(ek, B, os.urandom(32)))
            for k in range(1568):
                chiffré.append([Rc[k],Gc[k],Bc[k]])
            R=[A[i][j][0]]
            G=[A[i][j][1]]
            B=[A[i][j][2]]
```

chiffrer-img(A)

```python
    if l*c % 32 == 0 :
        Rc=bytearray(mk.k_pke_chiffrer(ek, R, os.urandom(32)))
        Gc=bytearray(mk.k_pke_chiffrer(ek, G, os.urandom(32)))
        Bc=bytearray(mk.k_pke_chiffrer(ek, B, os.urandom(32)))
        for k in range(1568):
            chiffré.append([Rc[k],Gc[k],Bc[k]])
        chiffré=np.array(chiffré,dtype='uint8')
        chiffré = chiffré.reshape((7*l,7*c,3))
        return l, c, chiffré
```

chiffrer-img(A)

```python
    else:
        while len(R)<32:
            R.append(0)
            G.append(0)
            B.append(0)
        Rc=bytearray(mk.k_pke_chiffrer(ek, R, os.urandom(32)))
        Gc=bytearray(mk.k_pke_chiffrer(ek, G, os.urandom(32)))
        Bc=bytearray(mk.k_pke_chiffrer(ek, B, os.urandom(32)))
        for k in range(1568):
            chiffré.append([Rc[k],Gc[k],Bc[k]])
        r = l*c % 32
        pixels_sup = 49*(32 - r)
        lignes_nécessaires  = int(np.ceil(pixels_sup / (7*c)))
        pixels_ajoutés = lignes_nécessaires*(7*c) - pixels_sup
        for i in range(pixels_ajoutés):
            chiffré.append([rd.randint(0,255),rd.randint(0,255),rd.randint(0,255)])
        chiffré=np.array(chiffré,dtype='uint8')
        total_lignes = 7*l + lignes_nécessaires
        chiffré = chiffré.reshape((total_lignes, 7*c, 3))
        return l, c, chiffré
```

déchiffrer-img(l, c, A)

```python
def déchiffrer_img(l, c, A):
    aplati = A.reshape(-1, 3)
    total_pixels = l * c
    nombre_blocs = int(np.ceil(total_pixels / 32))
    déchiffré = []
    for b in range(nombre_blocs):
        début = b * 1568
        fin   = début + 1568
        bloc = aplati[début:fin]
        Rc = [px[0] for px in bloc]
        Gc = [px[1] for px in bloc]
        Bc = [px[2] for px in bloc]
        R = bytearray(mk.k_pke_déchiffrer(dk, Rc))
        G = bytearray(mk.k_pke_déchiffrer(dk, Gc))
        B = bytearray(mk.k_pke_déchiffrer(dk, Bc))
        compte_pixels = 32
        if (b == nombre_blocs - 1) and (total_pixels % 32 != 0):
            compte_pixels = total_pixels % 32
        for k in range(compte_pixels):
            déchiffré.append([R[k], G[k], B[k]])
    aplati_déchiffré = np.array(déchiffré, dtype='uint8')
    image_déchiffrée  = aplati_déchiffré.reshape((l, c, 3))
    return image_déchiffrée
```

Affichage des images

```
img=plt.imread('/home/sharaf/MLKEM/img/Mosquee_Hassan_2.png')
img= 255*img
img=img.astype('uint8')
l, c, chiffré = chiffrer_img(img)
déchiffré = déchiffrer_img(l, c, chiffré)

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.title("Image originale")
plt.imshow(img)
plt.axis('off')

plt.subplot(1, 3, 2)
plt.title("Image chiffrée")
plt.imshow(chiffré, interpolation='nearest')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.title("Image déchiffrée")
plt.imshow(déchiffré)
plt.axis('off')

plt.show()
```