Estruturas criptograficas: TP3 problema 2

Neste trabalho pretende-se implementar em Sagemath um protótipo de um standard key-encapsulation mechanism ML-KEM derivado dos algoritmos KYBER parametrizado de acordo com as variantes sugeridas na norma (512, 768 e 1024 bits de segurança)

Especificamente, este padrão descreve primeiro um esquema de criptografia de chave pública denominado K-PKE e então usa os algoritmos de K-PKE como sub-rotinas ao descrever os algoritmos de ML-KEM . A transformação criptográfica de K-PKE para ML-KEM é crucial para alcançar a segurança total.

Um ML-KEM consiste nos algoritmos KeyGen, Encaps e Decaps, juntamente com uma coleção de conjuntos de parâmetros. No caso de ML-KEM, os três algoritmos mencionados acima são principais, sendo dependentes dos algoritmos PKEEncrypt, PKEDecrypt e PKEKeyGen.

ML-KEM

KeyGen

A chave de cifragem K-PKE servirá como a chave de encapsulamento de ML-KEM, a chave de decifragem juntamente com a de cifragem, o hash, e um randomness z, como chave de desencapsulamento.

Encaps

Primeiramente é cifrada uma mensagem aleatória m com 32 bytes, com a chave de encapsulamento.

O segredo compartilhado k e a randomness r são derivados da mensagem m e de um hash da chave de encapsulamento

A função retorna o segredo k e o cyphertext.

Decaps

Esta função recebe o cyphertext c, e a chave de desencapsulamento.

Numa primeira fase dá-se a extração dos parametros que compõem a chave de desencapsulamento (dk, ek H(ek), e z).

A chave de decifragem K-PKE é então utilizada para decifrar o cyphertext c e obter o plaintext m. O algoritmo recifra depois m e calcula uma shared key candidata K'.

Especificamente, K' e a aleatoriedade r' são calculados ao fazer o hash de m' e da chave de cifragem K-PKE, e um texto cifrado c' é gerado ao cifrar m' com a aleatoriedade r'.

Finalmente, é verificado se o texto cifrado resultante c' corresponde ao texto cifrado fornecido c. Se não corresponder, o algoritmo realiza uma "rejeição implícita": o valor de K' é alterado para um hash de c em conjunto com o valor aleatório z armazenado na chave secreta do ML-KEM. Em qualquer caso, o desencapsulamento produz a chave secreta partilhada resultante K'.

K-PKE

KeyGen

O processo começa gerando uma semente aleatória 'd'. Esta semente é então usada para produzir parâmetros aleatórios rho e sigma.

Em seguida, uma matriz quadrada aleatória A de tamanho k é gerada, e também um vetor 's' e 'e', compostos por k polinômios usando os parâmetros sigma e N.

Ambos os vetores 's' e 'e' são então transformados usando NTT.

O vetor t é obtido através da matriz A e dos vetores '_s' e '_e'.

Os elementos de t são codificados para formar a chave de cifragem ek_pke, à qual o valor rho é anexado. Para criar a chave de decifragem dk_pke o processo é semelhante mas com o vetor _s.

Este processo é essencial para garantir a segurança das comunicações criptográficas, fornecendo um método confiável para gerar chaves de criptografia pública no contexto do sistema K-PKE.

Encrypt

O método PKEEncrypt começa por extrair os parâmetros-chave da chave de encriptação 'ek', que incluem o vetor t e a semente ρ.

Em seguida, regenera a matriz A utilizando amostras NTT. O próximo passo é gerar vetores aleatórios r′, 'e1' e um termo de ruído 'e2'. Esses vetores são então transformados pela NTT para obter _r, que é usado no cálculo de 'u' e 'v'.

Finalmente, os vetores 'u' e 'v' são convertidos em bytes utilizando técnicas de codificação e compressão, e então retornados como o texto cifrado 'c'.

Decrypt

O texto cifrado c é dividido em duas partes c1 e c2 e posteriormente decodificados em u e v.

O segredo s é recuperado da chave de descriptografia dk_pke.

De seguida é calculado w usando v, s e NTT(u) para obter a mensagem.

A mensagem é convertida de volta para o formato original e retornada como plaintext.

In [54]:
```
import os
from hashlib import sha3_256, sha3_512, shake_128, shake_256
from functools import reduce
```

In [55]:
```
class PKE():

    def __init__(self, security_bits=512):

        if security_bits == 512:
            self.n = 256
            self.q = 3329
            self.k = 2
            self.eta1 = 3
            self.eta2 = 2
            self.du = 10
            self.dv = 4
        elif security_bits == 768:
            self.n = 256
            self.q = 3329
            self.k = 3
            self.eta1 = 2
            self.eta2 = 2
            self.du = 10
            self.dv = 4
        elif security_bits == 1024:
            self.n = 256
            self.q = 3329
            self.k = 4
            self.eta1 = 2
            self.eta2 = 2
            self.du = 11
            self.dv = 5


    def G(self, c):
        c_hash = sha3_512(c).digest()
        a = c_hash[:32]
        b = c_hash[32:]
        return a, b

    def H(self, m):
        return sha3_256(m).digest()

    def J(self, s):
        return shake_256(s).digest(32)

    def XOF(self, rho, i, j):
        temp = shake_128(rho + int.to_bytes(i) + int.to_bytes(j)).digest(
        return temp


    # Função NTT
    def NTT(self, f):
```

```
        f_ = list(f)

        k = 1
        len = 128
        while len >= 2:
            start = 0
            while start < 256:
                zeta = mod(17^(self.BitReverse(k)), self.q)
                k += 1
                for j in range(start, start + len):
                    t = mod(zeta * f_[j + len], self.q)
                    f_[j + len] = mod(f_[j] - t, self.q)
                    f_[j] = mod(f_[j] + t, self.q)



                start = start + 2 * len
            len = len // 2

        return f_

    # Função NTT Inversa
    def NTTInverse(self, f_):
        f = list(f_)

        k = 127
        len = 2
        while len <= 128:
            start = 0
            while start < 256:
                zeta = mod(17^(self.BitReverse(k)), self.q)
                k -= 1
                for j in range(start, start + len):
                    t = f[j]
                    f[j] = mod(t + f[j + len], self.q)
                    f[j + len] = mod(zeta * (f[j + len] - t), self.q)

                start = start + 2 * len
            len = len * 2

        for i in range(256):
            f[i] = mod(f[i] * 3303, self.q)

        return f

    # Retorna uma matriz de amostras NTT k*k*256
    def SampleNTT(self, B):
        i = 0
        j = 0
        a = [[0] for _ in range(256)]
        while j < 256:
            d1 = B[i] + 256 * (B[i + 1] % 16)
            d2 = floor(B[i + 1] / 16) + 16 * B[i + 2]
            if d1 < self.q:
                a[j] = d1
                j += 1
            if d2 < self.q and j < 256:
                a[j] = d2
                j += 1
            i += 3
```

```
        return a

    def PRF(self, s, b, eta):
        return shake_256(s + bytes(b)).digest(64 * eta)

    # Função auxiliar para transformar bytes em bits
    def BytesToBits(self, B):
        b = [0] * len(B) * 8
        B = self.BytesToByteArray(B)
        for i in range(len(B)):
            for j in range(0,8):
                b[8*i+j] = mod(B[i], 2)
                B[i] = B[i] // 2

        return b

    # Função auxiliar para transformar bits em bytes
    def BitsToBytes(self, b):
        l = len(b) // 8
        B = [0] * l
        for i in range(0,8*l):
            B[i // 8] += ZZ(b[i]) * 2^(mod(i,8))
        return B

    # Função auxiliar para transformar bytes em bytearray
    def ByteArrayToBytes(self, B):
        return bytes(B)

    # Função auxiliar para transformar bytearray em bytes
    def BytesToByteArray(self, Bytes):
        return list(Bytes)

    # Multiplicação de matrizes
    def MatrixMultiplication(self, A, u):
        aux = A.copy()
        res = [0] * self.n

        for i in range(self.k):
            aux[i] = self.MultiplyNTTs(A[i], u[i])

        for i in range(self.k):
            res = self.ArrayAddition(res, aux[i])

        return res

    # Adição de matrizes
    def MatrixAddition(self, A, B):
        res = []
        for i in range(self.k):
            res.append(self.ArrayAddition(A[i], B[i]))

        return res

    # Adição de vetores
    def ArrayAddition(self, A, B):
        res = [0] * self.n
        for i in range(self.n):
            res[i] = A[i] + B[i]
```

```
        return res

    # Subtração de vetores
    def ArraySubtraction(self, A, B):
        res = [0] * self.n
        for i in range(self.n):
            res[i] = A[i] - B[i]

        return res

    # Multiplicação de polinómios NTT
    def MultiplyNTTs(self, f, g):
        h = [0] * self.n
        for i in range(128):
            h[2*i], h[2*i + 1] = self.BaseCaseMultiply(f[2*i], f[2*i + 1]
        return h

    def BaseCaseMultiply(self, a0, a1, b0, b1, y):
        c0 = mod((a0 * b0) + (a1 * b1 * y), self.q)
        c1 = mod((a0 * b1) + (a1 * b0), self.q)
        return c0, c1

    def round(self, x):
        return int(x + 0.5)

    # Função auxiliar para inverter bits de um número com 7 bits
    def BitReverse(self, i):
        return int('{:07b}'.format(i)[::-1], 2)

    # Coeficientes de um polinómio de amostra
    def SamplePolyCBD(self, B, eta):
        b = self.BytesToBits(B)

        f = [0] * 256

        for i in range(256):
            x = 0
            y = 0
            for j in range(eta):
                x += b[2 * i * eta + j]
            for j in range(eta):
                y += b[2 * i * eta + eta + j]

            f[i] = mod((x - y), self.q)

        return f

    # Função auxiliar para codificar um polinómio de 256 coeficientes em
    def ByteEncode(self, F, d):

        if d < 12:
            m = 2^d
        elif d == 12:
            m = self.q

        b = [0] * (256 * d)
        for i in range(256):
            a = mod(F[i], 2^d)

            for j in range(d):
```

```
                b[i*d + j] = a % 2
                a = (ZZ(a) - ZZ(b[i*d + j])) / 2

        B = self.BitsToBytes(b)
        return B

    # Função auxiliar para decodificar um polinómio de 256 coeficientes a
    def ByteDecode(self, B, d):
        if d < 12:
            m = 2^d
        elif d == 12:
            m = self.q

        b = self.BytesToBits(B)
        F = [0] * 256
        for i in range(256):
            for j in range(0, d):
                F[i] += mod(ZZ(b[i*d + j]) * ZZ(2^j), m)

        return F

    # Função auxiliar de compressão
    def Compress(self, x, d):
        z = list(x)
        for i in range(len(x)):
            z[i] = mod(self.round((2^d) / self.q * ZZ(z[i])), 2^d)

        return z

    # Função auxiliar de descompressão
    def Decompress(self, y, d):
        z = list(y)
        for i in range(len(y)):
            z[i] = mod(self.round((self.q / 2^d) * ZZ(z[i])), self.q)

        return z

    # Geração de chave pública e privada
    def PKEKeyGen(self):
        d = os.urandom(32)
        rho, sigma = self.G(d)
        N = 0

        # Geração da matriz A k*k*256
        A = [[0] * self.k for _ in range(self.k)]
        for i in range(self.k):
            for j in range(self.k):
                A[i][j] = self.SampleNTT(self.XOF(rho, i, j))

        # Geração de s
        s = [[0] for _ in range(self.k)]
        for i in range(self.k):
            s[i] = self.SamplePolyCBD(self.PRF(sigma, N, self.eta1), self
            N += 1

        # Geração de e
        e = [[0] for _ in range(self.k)]
        for i in range(self.k):
            e[i] = self.SamplePolyCBD(self.PRF(sigma, N, self.eta1), self
            N += 1
```

```
        # Transformação de s e e em NTT
        _s, _e = [], []
        for i in range(self.k):
            _s.append(self.NTT(s[i]))
            _e.append(self.NTT(e[i]))

        # t = A º s + e
        t = [
                reduce(self.ArrayAddition, [
                self.MultiplyNTTs(A[j][i], _s[j])
                for j in range(self.k)
            ] + [_e[i]])
            for i in range(self.k)
        ]

        # Codificação de t e rho
        ek_pke = b''
        for i in range(self.k):
            ek_pke += self.ByteArrayToBytes(self.ByteEncode(t[i], 12))

        ek_pke += rho

        dk_pke = b''
        for i in range(self.k):
            dk_pke += self.ByteArrayToBytes(self.ByteEncode(_s[i], 12))

        return ek_pke, dk_pke

    # Função cifrar
    def PKEEncrypt(self, ek, m, r):
        N = 0

        # Transformação de ek em t e rho
        t = []
        for i in range(self.k):
            t.append(self.ByteDecode(self.BytesToByteArray(ek[i * 384:(i+

        rho = ek[self.k * 384:]

        # Geração da matriz A k*k*256 que foi usada para gerar a chave pú
        A = [[0] * self.k for _ in range(self.k)]
        for i in range(self.k):
            for j in range(self.k):
                A[i][j] = self.SampleNTT(self.XOF(rho, i, j))

        # Geração de r_
        r_ = [0] * self.k
        for i in range(self.k):
            r_[i] = self.SamplePolyCBD(self.PRF(r, N, self.eta1), self.et
            N += 1

        # Geração de e1 e e2
        e1 = [0] * self.k
        for i in range(self.k):
            e1[i] = self.SamplePolyCBD(self.PRF(r, N, self.eta2), self.et
            N += 1

        e2 = self.SamplePolyCBD(self.PRF(r, N, self.eta2), self.eta2)
```

```
        # Transformação de r_ em NTT
        _r = []
        for i in range(self.k):
            _r.append(self.NTT(r_[i]))

        # u = A º r_ + e1
        u = [
            self.ArrayAddition(self.NTTInverse(reduce(self.ArrayAddition,
                self.MultiplyNTTs(A[i][j], _r[j])
                for j in range(self.k)
            ])), e1[i])
            for i in range(self.k)
            ]

        mu2 = self.Decompress(self.ByteDecode(m, 1), 1)

        # v = t º r_ + e2 + mu2
        v = self.ArrayAddition(self.ArrayAddition(self.NTTInverse(self.Ma

        # Codificação de u e v
        c1 = b''
        for i in range(self.k):
            c1 += self.ByteArrayToBytes(self.ByteEncode(self.Compress(u[i

        c2 = self.ByteArrayToBytes(self.ByteEncode(self.Compress(v, self.

        return c1 + c2

    # Função decifrar
    def PKEDecrypt(self, dk_pke, c):

        # Extração de c1 e c2
        c1 = []
        for i in range(self.k):
            c1.append(self.BytesToByteArray(c[32 * i * self.du: 32 * (i +

        c2 = self.BytesToByteArray(c[32 * self.du * self.k: 32 * (self.du

        # Calcular u e v
        u = []
        for i in range(self.k):
            u.append(self.Decompress(self.ByteDecode(c1[i], self.du), sel

        v = self.Decompress(self.ByteDecode(c2, self.dv), self.dv)

        # Extração de s
        s = []
        for i in range(self.k):
            s.append(self.ByteDecode(dk_pke[i * 384: (i+1) * 384], 12))

        # Transformação de u em NTT
        for i in range(self.k):
            u[i] = self.NTT(u[i])

        # w = v - (s º u)
        w = self.MatrixMultiplication(s, u)
        w = self.ArraySubtraction(v, self.NTTInverse(w))

        # Codificação de w
```

```
        m = self.ByteEncode(self.Compress(w, 1), 1)

        return self.ByteArrayToBytes(m)
```

In [56]:
```
class ML_KEM():

    def __init__(self, bits=512):
        self.kem = PKE(bits)

    # Geração de chaves
    def keygen(self):
        z = os.urandom(32)

        ek_pke, dk_pke = self.kem.PKEKeyGen()

        ek = ek_pke
        dk = dk_pke + ek + self.kem.H(ek) + z

        return ek, dk

    # Encapsulamento
    def encaps(self, ek):
        m = os.urandom(32)

        # Segredo compartilhado K e randomness r
        K, r = self.kem.G(m + self.kem.H(ek))

        # Criptograma de m
        c = self.kem.PKEEncrypt(ek, m, r)

        return K, c

    # Desencapsulamento
    def decaps(self, c, dk):

        # Extração de dk_pke e ek_pke
        dk_pke = dk[:384 * self.kem.k]
        ek_pke = dk[384 * self.kem.k: 768 * self.kem.k + 32]

        # Extração de h (hash de ek_pke) e z (valor de rejeição)
        h = dk[768 * self.kem.k + 32: 768 * self.kem.k + 64]
        z = dk[768 * self.kem.k + 64: 768 * self.kem.k + 96]

        # decifragem de c
        m_ = self.kem.PKEDecrypt(dk_pke, c)

        K_, r_ = self.kem.G(m_ + h)
        K = self.kem.J(z + c)

        # "Re-cifrar" usando o r_ derivado
        c_ = self.kem.PKEEncrypt(ek_pke, m_, r_)

        # Verificação
        if c_ != c:
            print('Failed!')
            K_ = K
        else:
```

```
            print('Success!')

        return K_
```

Sucesso 512 bits

In [57]:
```
kem = ML_KEM(512)
ek, dk = kem.keygen()

K, c = kem.encaps(ek)

K_ = kem.decaps(c, dk)

print('Encapsuled:', K.hex())
print('Decapsuled:', K_.hex())
```

```
Success!
Encapsuled: 25aba98ebf9649d40b45007e32776c7d224d530d507d71c9298f88a01668b4
83
Decapsuled: 25aba98ebf9649d40b45007e32776c7d224d530d507d71c9298f88a01668b4
83
```

Rejeição 512 bits

In [58]:
```
kem = ML_KEM(512)
ek, dk = kem.keygen()

K, c = kem.encaps(ek)

# Alteração do último byte
c = c[:-1] + b'0'

K_ = kem.decaps(c, dk)

print('Encapsuled:', K.hex())
print('Decapsuled:', K_.hex())
```

```
Failed!
Encapsuled: e8692655baef65d53c1ad109a25e36dc81439aa3679db49d6761bccd18d1ed
dd
Decapsuled: 0ffce0ed7a16c36e120011de35143ed23ae6775f390ba92d4ef7e545a63b13
c4
```

Sucesso 756 bits

In [59]:
```
kem = ML_KEM(768)
ek, dk = kem.keygen()

K, c = kem.encaps(ek)

K_ = kem.decaps(c, dk)

print('Encapsuled:', K.hex())
print('Decapsuled:', K_.hex())
```

```
Success!
Encapsuled: 55a5ca7346f0daabf3584e33d16797ad5219e3cadb3f1011b1f7544cca565e
6d
Decapsuled: 55a5ca7346f0daabf3584e33d16797ad5219e3cadb3f1011b1f7544cca565e
6d
```

Sucesso 1024 bits

In [60]:
```
kem = ML_KEM(1024)
ek, dk = kem.keygen()

K, c = kem.encaps(ek)

K_ = kem.decaps(c, dk)

print('Encapsuled:', K.hex())
print('Decapsuled:', K_.hex())
```

```
Success!
Encapsuled: 847ad919508882a97ae548e4c1436d755681ad6acb8cfb0b87a360a8ae2b56
7b
Decapsuled: 847ad919508882a97ae548e4c1436d755681ad6acb8cfb0b87a360a8ae2b56
7b
```