Estruturas Criptográficas - TP3-2

PG53721 - Carlos Machado

PG54249 - Tiago Oliveira

Enunciado - Implementação KEM FIPS203

Neste problema, pretende-se implmentar um protótipo do standard parametrizado de um **Key Encapsulation Mechanism (KEM)** de acordo com as variantes sugeridas na norma **FIPS203** (512, 768 e 1024 bits de segurança).

In [30]:
```
import hashlib, os
from functools import reduce
```

Inicialização

In [31]:
```
def select_method(method_number):
    methods = {
        1: (2, 3, 10, 4),
        2: (3, 2, 10, 4),
        3: (4, 2, 11, 5)
    }
    return methods.get(method_number, None)

N = 256
Q = 3329
ETA2 = 2

print("Escolha um dos seguintes métodos:")
print("1. ML-KEM-512")
print("2. ML-KEM-768")
print("3. ML-KEM-1024")

method_number = int(input("Escreva o número correspondente ao método: "))

k_value, ETA1, DU, DV = select_method(method_number)
```

```
Escolha um dos seguintes métodos:
1. ML-KEM-512
2. ML-KEM-768
3. ML-KEM-1024
Escreva o número correspondente ao método: 1
```

Algoritmos auxiliares

Conversões e compressões

In [32]:
```
def bits_to_bytes(b):
    B = bytearray([0] * (len(b) // 8))
    for i in range(len(b)):
        B[i // 8] += b[i] * 2 ** (i % 8)
    return bytes(B)

def bytes_to_bits(B):
    B_list = list(B)
    b = [0] * (len(B_list) * 8)

    for i in range(len(B_list)):
        for j in range(8):
            b[8 * i + j] = B_list[i] % 2
            B_list[i] //= 2

    return b

def byte_encode(d, F):
    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 bits_to_bytes(b)

def byte_decode(d, B):
    m = 2 ** d if d < 12 else Q
    b = bytes_to_bits(B)
    F = [0] * 256
    for i in range(256):
        F[i] = sum(b[i * d + j] * (2 ** j) % m for j in range(d))

    return F

def compress(d, x):
    return [(((n * 2**d) + Q // 2 ) // Q) % (2**d) for n in x]

def decompress(d, x):
    return [(((n * Q) + 2**(d-1) ) // 2**d) % Q for n in x]
```

Sampling

In [33]:
```
def sample_ntt(B):
    a_nr = [0] * 256
    i = 0
    j = 0

    while j < 256:
        d1 = B[i] + 256 * (B[i + 1] % 16)
        d2 = (B[i + 1] // 16) + 16 * B[i + 2]

        if d1 < Q:
            a_nr[j] = d1
            j += 1

        if d2 < Q and j < 256:
            a_nr[j] = d2
            j += 1

        i += 3

    return a_nr

def sample_poly_cbd(B, eta):
    b = bytes_to_bits(B)
    f = [0] * 256

    for i in range(256):
        x = sum(b[2 * i * eta + j] for j in range(eta))
        y = sum(b[2 * i * eta + eta + j] for j in range(eta))
        f[i] = (x - y) % Q

    return f
```

NTT

In [34]:
```
def bit_rev_7(r):
    return int('{:07b}'.format(r)[::-1], 2)

ZETA = [pow(17, bit_rev_7(k_value), Q) for k_value in range(128)]
GAMMA = [pow(17, 2 * bit_rev_7(k_value) + 1, Q) for k_value in range(128)]

def ntt(f):
    fc = f
    k_value = 1
    l = 128

    while l >= 2:
        start = 0
        while start < 256:
            zeta = ZETA[k_value]
            k_value += 1
            for j in range(start, start + l):
                t = (zeta * fc[j + l]) % Q
                fc[j + l] = (fc[j] - t) % Q
                fc[j] = (fc[j] + t) % Q

            start += 2 * l

        l //= 2

    return fc

def ntt_inv(fc):
    f = fc
    k_value = 127
    l = 2
    while l <= 128:
        start = 0
        while start < 256:
            zeta = ZETA[k_value]
            k_value -= 1
            for j in range(start, start + l):
                t = f[j]
                f[j] = (t + f[j + l]) % Q
                f[j + l] = (zeta * (f[j + l] - t)) % Q

            start += 2 * l

        l *= 2

    return [(felem * 3303) % Q for felem in f]


def base_case_multiply(a0, a1, b0, b1, gamma):
    c0 = a0 * b0 + a1 * b1 * gamma
    c1 = a0 * b1 + a1 * b0

    return c0, c1

def multiply_ntt_s(fc, gc):
    hc = [0] * 256
    for i in range(128):
        hc[2 * i], hc[2 * i + 1] = base_case_multiply(fc[2 * i], fc[2 * i +
```

```
    return hc
```

Funções auxiliares

In [35]:
```
def XOF(rho, i, j):
    return hashlib.shake_128(rho + bytes([i]) + bytes([j])).digest(1500)

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

def vector_add(ac, bc):
    return [(x + y) % Q for x, y in zip(ac, bc)]

def vector_sub(ac, bc):
    return [(x - y) % Q for x, y in zip(ac, bc)]

def G(c):
    G_result = hashlib.sha3_512(c).digest()
    return G_result[:32], G_result[32:]

def H(c):
    return hashlib.sha3_256(c).digest()

def J(s, l):
    return hashlib.shake_256(s).digest(l)
```

K-PKE

In [36]:

```
def k_pke_keygen():
    d = os.urandom(32)
    rho, sigma = G(d)
    N = 0
    Ac = []

    for i in range(k_value):
        row = []
        for j in range(k_value):
            row.append(sample_ntt(XOF(rho, i, j)))
        Ac.append(row)

    s = []
    for i in range(k_value):
        s.append(sample_poly_cbd(PRF(ETA1, sigma, bytes([N])), ETA1))
        N += 1

    e = []
    for i in range(k_value):
        e.append(sample_poly_cbd(PRF(ETA1, sigma, bytes([N])), ETA1))
        N += 1

    sc = [ntt(s[i]) for i in range(k_value)]
    ec = [ntt(e[i]) for i in range(k_value)]
    tc = [reduce(vector_add, [multiply_ntt_s(Ac[i][j], sc[j]) for j in range

    ek_PKE = b"".join(byte_encode(12, tc_elem) for tc_elem in tc) + rho
    dk_PKE = b"".join(byte_encode(12, sc_elem) for sc_elem in sc)

    return ek_PKE, dk_PKE

def k_pke_encrypt(ek_PKE, m, rand):
    N = 0
    tc = [byte_decode(12, ek_PKE[i * 384 : (i + 1) * 384]) for i in range(k_
    rho = ek_PKE[384 * k_value : 384 * k_value + 32]

    Ac = []
    for i in range(k_value):
        row = []
        for j in range(k_value):
            row.append(sample_ntt(XOF(rho, i, j)))
        Ac.append(row)

    r = []
    for i in range(k_value):
        r.append(sample_poly_cbd(PRF(ETA1, rand, bytes([N])), ETA1))
        N += 1

    e1 = []
    for i in range(k_value):
        e1.append(sample_poly_cbd(PRF(ETA2, rand, bytes([N])), ETA2))
        N += 1

    e2 = sample_poly_cbd(PRF(ETA2, rand, bytes([N])), ETA2)
    rc = [ntt(r[i]) for i in range(k_value)]

    u = [vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(Ac[j][i], rc[

    mu = decompress(1, byte_decode(1, m))
```

```
    mu = decompress(1, byte_decode(1, m))

    v = vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(tc[i], rc[i])

    c1 = b"".join(byte_encode(DU, compress(DU, u[i])) for i in range(k_value
    c2 = byte_encode(DV, compress(DV, v))

    return c1 + c2

def k_pke_decrypt(dk_PKE, c):
    c1 = c[:32 * DU * k_value]
    c2 = c[32 * DU * k_value : 32 * (DU * k_value + DV)]
    u = [decompress(DU, byte_decode(DU, c1[i * 32 * DU : (i + 1) * 32 * DU])

    v = decompress(DV, byte_decode(DV, c2))

    sc = [byte_decode(12, dk_PKE[i * 384 : (i + 1) * 384]) for i in range(k_

    w = vector_sub(v, ntt_inv(reduce(vector_add, [multiply_ntt_s(sc[i], ntt(

    return byte_encode(1, compress(1, w))
```

L-KEM Key-Encapsulation Mechanism

In [37]:
```python
def ml_kem_keygen():
    z = os.urandom(32)
    ek_PKE, dk_PKE = k_pke_keygen()
    ek = ek_PKE
    dk = dk_PKE + ek + H(ek) + z

    return ek, dk


def ml_kem_encaps(ek):

    # Input validation

    # Type check
    assert(len(ek) == 384*k_value+32)
    # Modulus check
    #ek_aux = byte_encode(12,byte_decode(12,ek))
    #assert(ek == ek_aux)

    m = os.urandom(32)
    K, r = G(m + H(ek))
    c = k_pke_encrypt(ek, m, r)

    return K, c


def ml_kem_decaps(c, dk):

    # Input validation

    # Cipher text typecheck
    assert(len(c) == 32*(DU*k_value+DV))
    # Decapsulation key typecheck
    assert(len(dk) == 768*k_value+96)

    dk_PKE = dk[0: 384 * k_value]
    ek_PKE = dk[384 * k_value : 768 * k_value + 32]
    h = dk[768 * k_value + 32 : 768 * k_value + 64]
    z = dk[768 * k_value + 64 : 768 * k_value + 96]
    ml = k_pke_decrypt(dk_PKE, c)
    Kl, rl = G(ml + h)
    Kb = J((z + c), 32)
    cl = k_pke_encrypt(ek_PKE, ml, rl)
    if c != cl:
        Kl = Kb

    return Kl
```

Teste

In [38]:
```
rand = os.urandom(32)
print('Teste de cifragem K-PKE\n\n')
message = b'Este e um exemplo de mensagem !!'
ek_PKE, dk_PKE = k_pke_keygen()
#
print('message:', message)
print('\n')
cipher = k_pke_encrypt(ek_PKE, message, rand)
#print(cipher)
#print('\n')
#
message2 = k_pke_decrypt(dk_PKE, cipher)
print('recovered message:', message2)
```

```
Teste de cifragem K-PKE


message: b'Este e um exemplo de mensagem !!'


recovered message: b'Este e um exemplo de mensagem !!'
```

In [39]:
```
print('Teste ML-KEM\n')
ek, dk = ml_kem_keygen()
#print('ek:', ek)
#print('dk:', dk)

K, c = ml_kem_encaps(ek)
#print('K:', K)
#print('c:', c)

K1 = ml_kem_decaps(c, dk)

if K == K1:
    print('Teste passou com sucesso. As chaves são iguais!')
else:
    print('Teste falhou')
```

```
Teste ML-KEM

Teste passou com sucesso. As chaves são iguais!
```

In [ ]:

In [ ]: