

Estruturas Cripográficas - Criptografia e Segurança da Informação

TP3 - Exercício 2 - KEM

Tratado realizado por:

Número	Nome
PG54177	Ricardo Alves Oliveira
PG54236	Simão Oliveira Alvim Barroso

Enunciado

2. Em Agosto de 2023 a NIST publicou um draf da [norma FIPS203](#) para um Key Encapsulation Mechanism (KEM) derivado dos algoritmos [KYBER](#). O preâmbulo do “draft”

A key-encapsulation mechanism (or KEM) is a set of algorithms that, under certain conditions, can be used by two parties to establish a shared secret key over a public channel. A shared secret key that is securely established using a KEM can then be used with symmetric-key cryptographic algorithms to perform basic tasks in secure communications, such as encryption and authentication. This standard specifies a key-encapsulation mechanism called ML-KEM. The security of ML-KEM is related to the computational difficulty of the so-called Module Learning with Errors problem. At present, ML-KEM is believed to be secure even against adversaries who possess a quantum computer

Neste trabalho pretende-se implementar em Sagemath um protótipo deste standard parametrizado de acordo com as variantes sugeridas na norma (512, 768 e 1024 bits de segurança)

Imports

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

Inicialização

Dependendo das variantes sugeridas na norma, existem diferentes valores para os diferentes parâmetros, como vemos na seguinte imagem retirada da norma FIPS203:

	n	q	k	η_1	η_2	d_u	d_v	required RBG strength (bits)
ML-KEM-512	256	3329	2	3	2	10	4	128
ML-KEM-768	256	3329	3	2	2	10	4	192
ML-KEM-1024	256	3329	4	2	2	11	5	256

```
In [ ]: sec_bits = int(input("n value: "))

assert sec_bits in [512,768,1024], "sec_bits must be 512, 768 or 1024"

N = 256
Q = 3329

if sec_bits == 512:
    k_val=2
    eta1=3
    eta2=2
    DU=10
    DV=5

elif sec_bits == 768:
    k_val=3
    eta1=2
    eta2=2
    DU=10
    DV=4

elif sec_bits == 1024:
    k_val=4
    eta1=2
    eta2=2
    DU=11
    DV=5

print(f"ML-KEM-{{sec_bits}} ")
```

ML-KEM-1024

O algoritmo ML-KEM é composto por 3 algoritmos:

1. Key generation (ML-KEM.KeyGen)
2. Encapsulation (ML-KEM.Encaps)
3. Decapsulation (ML-KEM.Decaps)

Cada um desses algoritmos é compostos por diferentes sub-algoritmos, que são descritos na norma FIPS203. Ao longo das seguintes células vamos implementar cada um desses algoritmos.

List of Algorithms

Algorithm 1	ForExample	7
Algorithm 2	BitsToBytes(b)	17
Algorithm 3	BytesToBits(B)	18
Algorithm 4	ByteEncode _{d} (F)	19
Algorithm 5	ByteDecode _{d} (B)	19
Algorithm 6	SampleNTT(B)	20
Algorithm 7	SamplePolyCBD _{η} (B)	20
Algorithm 8	NTT(f)	22
Algorithm 9	NTT ⁻¹ (\hat{f})	23
Algorithm 10	MultiplyNTTs(\hat{f}, \hat{g})	24
Algorithm 11	BaseCaseMultiply($a_0, a_1, b_0, b_1, \gamma$)	24
Algorithm 12	K-PKE.KeyGen()	26
Algorithm 13	K-PKE.Encrypt(ek_{PKE}, m, r)	27
Algorithm 14	K-PKE.Decrypt(dk_{PKE}, c)	28
Algorithm 15	ML-KEM.KeyGen()	29
Algorithm 16	ML-KEM.Encaps(ek)	30
Algorithm 17	ML-KEM.Decaps(c, dk)	32

Nestas celulas em baixo temos alguns algoritmos auxiliares aos algoritmos utilizado no ML-KEM.

- bit_rev_7: Função encarregada de reorganizar os bits de um número inteiro de sete bits.
- G: Função que utiliza o algoritmo SHA3-512 para gerar dois resultados, cada um com trinta e dois bytes, a partir de uma entrada de tamanho variável em bytes.
- H: Função que utiliza o algoritmo SHA3-256 para gerar um resultado com trinta e dois bytes a partir de uma entrada de tamanho variável em bytes.
- J: Função que utiliza o algoritmo SHAKE256 para produzir um resultado com trinta e dois bytes a partir de uma entrada de tamanho variável em bytes.
- XOF: Função de saída extensível que utiliza o algoritmo SHAKE128. Recebe como entrada uma variável de trinta e dois bytes e duas variáveis de um byte cada, produzindo um resultado em bytes de tamanho variável.
- PRF: Função que utiliza o algoritmo SHAKE256 para gerar um resultado em bytes.
- vector_add: Algoritmo responsável por realizar a adição de dois vetores (módulo q).
- vector_sub: Algoritmo responsável por realizar a subtração de dois vetores (módulo q).
- compress: Função capaz de reduzir a quantidade de informação em um vetor de valores inteiros.
- decompress: Função responsável por restaurar os valores inteiros de um vetor que foi comprimido.

Agora, vamos às variáveis globais utilizadas:

- **ZETA**: variável utilizada nas funções **ntt** e **ntt_inv**, obtida através da exponenciação da raiz primitiva $\zeta = 17$
- **GAMMA**: variável utilizada na função **multiply_ntt_s**, obtida através da exponenciação da raiz primitiva $\zeta = 17$

```

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

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)

def XOF(rho, i, j):
    return hashlib.shake_128(rho + bytes([i]) + bytes([j])).digest(1536)

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 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]

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

```

Temos aqui em baixo a codificação dos algoritmos 2,3,4 e 5 presentes na lista de algoritmos que está presente no FIPS 203.

```

In [ ]: 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

```

Temos aqui em baixo a codificação dos algoritmos 6 e 7 presentes na lista de algoritmos que está presente no FIPS 203.

```

In [ ]: def sample_ntt(B):
        i, j = 0, 0
        ac = [0] * 256

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

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

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

            i += 3

        return ac

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

```

Temos aqui em baixo a codificação dos algoritmos 8, 9, 10 e 11 presentes na lista de

algoritmos que está presente no FIPS 203.

```
In [ ]: def ntt(f):
    fc = f
    k = 1
    len = 128

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

            start += 2 * len

        len //= 2

    return fc

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

            start += 2 * len

        len *= 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 + 1], gc[2 * i], gc[2 * i + 1], gamma)

    return hc
```

Nas 3 células abaixo temos os algoritmos 12, 13 e 14.

```
In [ ]: def k_pke_keygen(k, eta1):
    d = os.urandom(32)
    rho, sigma = G(d)
    N = 0
    Ac = [[None for _ in range(k)] for _ in range(k)]
    s = [None for _ in range(k)]
    e = [None for _ in range(k)]

    for i in range(k):
        for j in range(k):
            Ac[i][j] = sample_ntt(XOF(rho, i, j))

    for i in range(k):
        s[i] = sample_poly_cbd(PRF(eta1, sigma, bytes([N])), eta1)
        N += 1

    for i in range(k):
        e[i] = sample_poly_cbd(PRF(eta1, sigma, bytes([N])), eta1)
        N += 1

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

    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
```

```
In [ ]: def k_pke_encrypt(ek_PKE, m, rand, k, eta1, eta2, du, dv):
    N = 0
    tc = [byte_decode(12, ek_PKE[i * 384 : (i + 1) * 384]) for i in range(k)]
    rho = ek_PKE[384 * k : 384 * k + 32]
    Ac = [[None for _ in range(k)] for _ in range(k)]
    r = [None for _ in range(k)]
    e1 = [None for _ in range(k)]

    for i in range(k):
        for j in range(k):
            Ac[i][j] = sample_ntt(XOF(rho, i, j))

    for i in range(k):
        r[i] = sample_poly_cbd(PRF(eta1, rand, bytes([N])), eta1)
        N += 1

    for i in range(k):
        e1[i] = 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)]
    u = [vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(Ac[j][i], tc[j]) for j in range(k)])),
                    mu = decompress(1, byte_decode(1, m)))
    v = vector_add(ntt_inv(reduce(vector_add, [multiply_ntt_s(tc[i], rc[j]) for j in range(k)])),
                    mu = decompress(1, byte_decode(1, m)))

    c1 = b"".join(byte_encode(du, compress(du, u[i])) for i in range(k))
    c2 = byte_encode(dv, compress(dv, v))
```

```
return c1 + c2
```

```
In [ ]: def k_pke_decrypt(dk_PKE, c, k, du, dv):
    c1 = c[:32 * du * k]
    c2 = c[32 * du * k : 32 * (du * k + dv)]
    u = [decompress(du, byte_decode(du, c1[i * 32 * du : (i + 1) * 32 * d
    v = decompress(dv, byte_decode(dv, c2))]
    sc = [byte_decode(12, dk_PKE[i * 384 : (i + 1) * 384])] for i in range
    w = vector_sub(v, ntt_inv(reduce(vector_add, [multiply_ntt_s(sc[i], n

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

Temos agora aqui os 3 ultimos algoritmos necessários para a implementação do ML-KEM.

```
In [ ]: def ml_kem_keygen(k, eta1):
    z = os.urandom(32)
    ek_PKE, dk_PKE = k_pke_keygen(k, eta1)
    ek = ek_PKE
    dk = dk_PKE + ek + H(ek) + z

    return ek, dk

def ml_kem_encaps(ek, k, eta1, eta2, du, dv):
    m = os.urandom(32)
    K, r = G(m + H(ek))
    c = k_pke_encrypt(ek, m, r, k, eta1, eta2, du, dv)

    return K, c

def ml_kem_decaps(c, dk, k, eta1, eta2, du, dv):
    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]
    ml = k_pke_decrypt(dk_PKE, c, k, du, dv)
    Kl, rl = G(ml + h)
    Kb = J((z + c), 32)
    cl = k_pke_encrypt(ek_PKE, ml, rl, k, eta1, eta2, du, dv)
    if c != cl:
        Kl = Kb

    return Kl
```

Temos agora aqui uma função para testar o ML-KEM.

```
In [ ]: def test_function(k, eta1, eta2, du, dv):
    ek, dk = ml_kem_keygen(k, eta1)

    assert type(ek) == bytes, "Type check failed for ek"
    assert len(ek) == 384 * k + 32, "Length check failed for ek"

    K, c = ml_kem_encaps(ek, k, eta1, eta2, du, dv)

    assert type(c) == bytes, "Type check failed for ek"
```



```
assert len(c) == 32 * (du * k + dv), "Length check failed for ek"

Kl = ml_kem_decaps(c, dk, k, eta1, eta2, du, dv)

print('Chaves iguais?', K == Kl)
```

```
In [ ]: print(f"ML-KEM-{{sec_bits}}")

        test_function(k_val, eta1, eta2, DU, DV)
```

ML-KEM-1024

Chaves iguais? True