Estruturas Criptográficas

Trabalho Prático 3 - Exercício 2

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

Em agosto de 2023, a **NIST** publicou um *draft* da norma **FIPS203** para um **Key Encapsulation Mechanism** (KEM) derivado dos algoritmos **KYBER**.

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).

Resolução

Em primeiro lugar, mostram-se os *imports* necessários.

In [1]:
```
import hashlib, os
from functools import reduce
```

Funções auxiliares e variáveis globais

Em primeiro lugar, apresentam-se as funções auxiliares e as variáveis globais utilizadas na realização do projeto proposto. Em primeiro lugar, as funções auxiliares apresentam-se como porções de código que, ao longo do projeto, se mostraram úteis quando aplicadas com as funções principais do protótipo. São as seguintes:

- **bit_rev_7**: função responsável por transformar um valor inteiro, de sete *bits*, trocando a ordem dos seus *bits*
- **G**: função que, recorrendo ao **SHA3-512**, produz dois *outputs*, de trinta e dois *bytes* cada, através de um *input*, em *bytes*, de tamanho variável
- **H**: função que, recorrendo ao **SHA3-256**, produz um *output*, de trinta e dois *bytes*, a partir de um *input* de *bytes*, de tamanho variável
- **J**: função que, recorrendo ao **SHAKE256**, produz um *output*, de trinta e dois *bytes*, a partir de um *input* de *bytes*, de tamanho variável

- **XOF**: **eXtendable-output function** que, recorrendo ao **SHAKE128** e, tendo como *inputs* uma variável de trinta e dois *bytes* e duas variáveis de um *byte* cada, produz um *output*, em *bytes*, de tamanho variável
- **PRF**: função pseudoaleatória que, através do **SHAKE256**, produz um *output* em *bytes*
- **vector_add**: algoritmo responsável por fazer a soma de dois vetores (módulo q)
- **vector_sub**: algoritmo responsável por executar a subtração de dois vetores (módulo q)
- **compress**: função com a capacidade de comprimir os valores inteiros de um vetor
- **decompress**: função responsável por descomprimir os valores inteiros de um vetor

Em segundo lugar, apresentam-se as variáveis globais utilizadas no protótipo:

- **Q**: inteiro primo, obtido através de **$2^8 * 13 + 1$**
- **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 [2]:
```
Q = 3329


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)]
```

Algoritmos

Aqui, apresentam-se os dezasseis algoritmos que formam o corpo principal do protótipo estudado e desenvolvido. Segue-se uma explicação breve de cada um dos mesmos:

- **bits_to_bytes**: função responsável por converter um *array* de *bits* numa representação em *bytes*
- **bytes_to_bits**: função com a capacidade de converter um *array* de *bytes* num *array* de *bits*
- **byte_encode**: algoritmo capaz de converter um *array* de *bits* (representados por inteiros) num *array* de *bytes*
- **byte_decode**: função que decodifica um *array* de *bytes*, transformando-o numa representação em *bits* (inteiros)
- **sample_ntt**: algoritmo que converte uma *stream* de *bytes* num polinómio no domínio *NTT*
- **sample_poly_cbd**: algoritmo que produz uma amostra aleatória da distribuição Dη(Rq)
- **ntt**: algoritmo que calcula a representação **NTT** de um dado polinómio
- **ntt_inv**: função que calcula um polinómio, através da sua representação **NTT**
- **base_case_multiply**: algoritmo que computa o produto de dois polinómios, de grau um, em relação a um módulo quadrático
- **multiply_ntt_s**: função responsável por calcular o produto, no anel $\mathbb{T}_q$, de duas representações **NTT**
- **k_pke_keygen**: algoritmo que gera uma chave de cifragem e a respetiva chave de decifragem
- **k_pke_encrypt**: algoritmo que, através de uma chave de cifragem gerada e de um valor aleatório, cifra uma mensagem de texto

- **k_pke_decrypt**: algoritmo que, recorrendo a uma chave de decifragem gerada, decifra um texto cifrado
- **ml_kem_keygen**: função que gera chaves de encapsulamento e desencapsulamento
- **ml_kem_encaps**: função que, através de uma chave de encapsulamento, gera uma chave partilhada e um texto cifrado associado
- **ml_kem_decaps**: função que, a partir de uma chave de desencapsulamento e de um texto cifrado, gera uma chave partilhada

In [3]:
```
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 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


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 +

    return hc


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

    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, 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], rc[
    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))
    c2 = byte_encode(dv, compress(dv, v))

    return c1 + c2


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 * 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))


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
```

Testes de aplicação

Para efeitos de teste, desenvolveu-se a função **ml_kem_test**, responsável por receber os diversos parâmetros do mecanismo **ML-KEM**, de acordo com o nível de segurança e a performance pretendidos.

In [4]:
```
def ml_kem_test(k, eta1, eta2, du, dv):
    ek, dk = ml_kem_keygen(k, eta1)

    if type(ek) != bytes or len(ek) != 384 * k + 32:
        raise ValueError('invalid ek (type check)')

    if b''.join([byte_encode(12, decoded_ek_elem) for decoded_ek_elem in [by
        raise ValueError('invalid dk (type check)')

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

    if type(c) != bytes or len(c) != 32 * (du * k + dv):
        raise ValueError('invalid c (type check)')

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

    print('Equal shared keys?', K == Kl)
```

ML-KEM-512

In [5]:
```
ml_kem_test(2, 3, 2, 10, 4)
```

```
Equal shared keys? True
```

ML-KEM-768

In [6]:
```
ml_kem_test(3, 2, 2, 10, 4)
```

```
Equal shared keys? True
```

ML-KEM-1024

In [7]:
```
ml_kem_test(4, 2, 2, 11, 5)
```

```
Equal shared keys? True
```