Estruturas Criptográficas

Trabalho Prático 4 - Exercício 1

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

Implemente um protótipo do esquema descrito no *draft* **FIPS 204**, que deriva do algoritmo **Dilithium**.

Resolução

Em primeiro lugar, apresentam-se os *imports* efetuados.

In [25]:
```
from functools import reduce
import copy, hashlib, math, random
```

Código auxiliar

Em primeiro lugar, explicam-se as diversas funções auxiliares que, ao longo de todo o código, se mostraram muito importantes. Explica-se, também, a variábel global utilizada em conjunto com os diversos algoritmos implementados (que serão descritos e explicados de seguida):

- **Q**: número primo igual a 8380417
- **H**: função **XOF** que recorre ao **SHAKE256**
- **H128**: função **XOF** que recorre ao **SHAKE128**
- **jth_byte**: algoritmo que extrai um *byte*, representado por uma *bit string*, após aplicação de uma das funções **XOF**
- **mod_plus_minus**: função que devolve um valor, num intervalo condicionado por um dos argumentos, de forma a que esse mesmo valor e o outro argumento sejam congruentes módulo o primeiro (argumento referido)
- **brv**: função "*bit reversal*"
- **vector_add**: algoritmo que efetua a soma de dois **NTTs**
- **vector_sub**: algoritmo que efetua a subtração de dois **NTTs**
- **vector_mult**: algoritmo que efetua a multiplicação de dois **NTTs**

- **matrix_vector_mult**: função que efetua a multiplicação entre uma matriz e um vetor, em $\mathbb{T}_q$
- **infinity_norm**: algoritmo que representa a "norma infinito"

In [26]:
```
Q = 8380417

def H(v, d):
    H_object = hashlib.shake_256(bytes(v)).digest(d // 8)
    return [int(bit) for byte in H_object for bit in f'{byte:08b}']


def H128(v, d):
    H_128_object = hashlib.shake_128(bytes(v)).digest(d // 8)
    return [int(bit) for byte in H_128_object for bit in f'{byte:08b}']


def jth_byte(rho, j, hash):
    hash_object = hash(rho, 8 * (j + 1))
    hash_object_bits = [int(bit) for byte in hash_object for bit in f'{byte:

    return hash_object_bits[8 * j : 8 * j + 8]


def mod_plus_minus(m, alpha):
    if alpha % 2 == 0:
        lim = alpha // 2

    else:
         lim = (alpha - 1) // 2

    mod = m % alpha
    if mod > lim:
        mod -= alpha

    return mod


def brv(r):
    brv_r = 0
    for i in range(8):
        brv_r |= ((r >> i) & 1) << (7 - i)

    return brv_r


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 vector_mult(ac, bc):
    return [(x * y) % Q for x, y in zip(ac, bc)]
```

```python
def matrix_vector_mult(Ac, bc):
    result = []
    for i in range(len(bc)):
        mid_result = []
        for j in range(len(Ac[i])):
            mid_result.append(vector_mult(Ac[i][j], bc[j]))
        result.append(reduce(vector_add, mid_result))

    return result


def infinity_norm(matrix):
    aux = []
    for vector in matrix:
        aux_vector = []
        for elem in vector:
            aux_vector.append(mod_plus_minus(elem, Q))

        aux.append(max(aux_vector))

    return max(aux)
```

Algoritmos

Aqui, apresentam-se todos os algoritmos desenvolvidos que, no fundo, representam a íntegra do protótipo desenvolvido e que foram, na totalidade, escritos dentro de uma classe (**MLDSA**). Refere-se, ainda, que a inicialização da classe se efetua de acordo com os parâmetros regulamentados. Deste modo, explica-se, de uma forma simples, a principal função de cada um dos mesmos:

- **ml_dsa_keygen**: algoritmo que gera um par de chaves, pública e privada
- **ml_dsa_sign**: função que gera uma assinatura (não desenvolvida)
- **ml_dsa_verify**: função que verifica uma assinatura (não desenvolvida)
- **integer_to_bits**: algoritmo que produz uma *bit string*, a partir de um inteiro, em *little-endian*
- **bits_to_integer**: função que computa um inteiro, através da sua representação em *bit string*, em *little-endian*
- **bits_to_bytes**: algoritmo que converte uma *bit string* numa *byte string*
- **bytes_to_bits**: algoritmo que converte uma *byte string* numa *bit string*
- **coeff_from_three_bytes**: função que gera um inteiro, não negativo, módulo **q**
- **coeff_from_half_byte**: algoritmo que gera um inteiro num intervalo específico
- **simple_bit_pack**: função que codifica um polinómio numa *byte string*
- **bit_pack**: algoritmo que codifica um polinómio numa *byte string*

- **simple_bit_unpack**: função que reverte o procedimento da **simple_bit_pack**
- **bit_unpack**: algoritmo que reverte o procedimento da **bit_pack**
- **hint_bit_pack**: função que codifica um polinómio constituído por coeficientes binários numa *byte string*
- **hint_bit_unpack**: função que reverte o procedimento da **hint_bit_pack**
- **pk_encode**: algoritmo que codifica uma chave pública numa *byte string*
- **pk_decode**: algoritmo que reverte o procedimento da **pk_encode**
- **sk_encode**: algoritmo que codifica uma chave privada numa *byte string*
- **sk_decode**: algoritmo que reverte o procedimento da **sk_encode**
- **sig_encode**: função que codifica uma assinatura numa *byte string*
- **sig_decode**: função que reverte o procedimento da **sig_encode**
- **w1_encode**: algoritmo que codifica um vetor polinomial numa *bit string*
- **sample_in_ball**: algoritmo que produz um polinómio com coeficientes em {-1, 0, 1} e com um peso de **Hamming** definido
- **rej_ntt_poly**: função que produz um polinómio em $\mathbb{T}_q$
- **rej_bounded_poly**: função que produz um polinómio em $\mathbb{R}_q$, com os coeficientes num intervalo específico
- **expand_a**: algoritmo que produz uma matriz de elementos de $\mathbb{T}_q$
- **expand_s**: função que produz dois vetores com os coeficientes num intervalo específico
- **expand_mask**: algoritmo que produz um vetor, de forma a que os seus elementos possuam os seus coeficientes num intervalo reservado
- **power_2_round**: função que, a partir de um valor **r**, produz dois valores, **r0** e **r1**, que obedecem à regra $\mathbf{r \equiv r_1 2^d + r_0 \bmod q}$
- **decompose**: função que, a partir de um valor **r**, produz dois valores, **r0** e **r1**, que obedecem à regra $\mathbf{r \equiv r_1(2\gamma_2) + r_0 \bmod q}$
- **high_bits**: algoritmo que devolve a primeira componente do *output* da função **decompose**
- **low_bits**: algoritmo que devolve a segunda componente do *output* da função **decompose**
- **make_hint**: função que verifica se adicionar um valor a outro altera os *high bits* do segundo
- **use_hint**: algoritmo que devolve os *high bits* de um valor, ajustados de acordo com o *hint* de outro valor
- **ntt**: algoritmo que computa a **Number-Theoretic Transform**
- **ntt_inv**: algoritmo que reverte a **Number-Theoretic Transform**

In [27]:
```
class MLDSA():

    def __init__(self, tau, lmbda, gamma1, gamma2_div, k, l, eta, omega):
        self.d = 13
        self.tau = tau
```

```
        self.lmbda = lmbda
        self.gamma1 = gamma1
        self.gamma2 = (Q - 1) // gamma2_div
        self.k, self.l = k, l
        self.eta = eta
        self.beta = self.tau * self.eta
        self.omega = omega
        self.zeta = 1753



    def ml_dsa_keygen(self):
        csi = [random.randint(0, 1) for _ in range(256)]
        H_csi = H(csi, 1024)
        rho, rhol, K = H_csi[:256], H_csi[256 : 768], H_csi[768:]

        Ac = self.expand_a(rho)
        s1, s2 = self.expand_s(rhol)

        ntt_s1 = [self.ntt(s1_elem) for s1_elem in s1]
        Ac_ntt_s1 = matrix_vector_mult(Ac, ntt_s1)
        ntt_inv_Ac_ntt_s1 = [self.ntt_inv(Ac_ntt_s1_elem) for Ac_ntt_s1_elem

        t = [vector_add(ntt_inv_Ac_ntt_s1[i], s2[i]) for i in range(len(ntt_
        for i in range(len(ntt_inv_Ac_ntt_s1), len(s2)):
            t.append(s2[i])

        t1, t0 = [], []
        for vector in t:
            r0_vector = []
            r1_vector = []
            for r in vector:
                r1, r0 = self.power_2_round(r)
                r0_vector.append(r0)
                r1_vector.append(r1)

            t1.append(r1_vector)
            t0.append(r0_vector)

        pk = self.pk_encode(rho, t1)
        tr = H(self.bytes_to_bits(pk), 512)
        sk = self.sk_encode(rho, K, tr, s1, s2, t0)

        return pk, sk


    def ml_dsa_sign(self, sk, M):
        pass


    def ml_dsa_verify(self, pk, M, sigma):
        pass


    def integer_to_bits(self, x, alpha):
        y = [None for _ in range(alpha)]
```

```
        for i in range(alpha):
            y[i] = x % 2
            x //= 2

        return y


    def bits_to_integer(self, y, alpha):
        x = 0
        for i in range(1, alpha + 1):
            x = 2 * x + y[alpha - i]

        return x


    def bits_to_bytes(self, y):
        c = len(y)
        z_len = math.ceil(c // 8)
        z = [0 for _ in range(z_len)]

        for i in range(c):
            z[i // 8] += y[i] * (2 ** (i % 8))

        return z


    def bytes_to_bits(self, z):
        zz = copy.deepcopy(z)
        d = len(zz)
        y = [0 for _ in range(d * 8)]

        for i in range(d):
            for j in range(8):
                y[8 * i + j] = zz[i] % 2
                zz[i] //= 2

        return y


    def coeff_from_three_bytes(self, b0, b1, b2):
        if b2 > 127:
            b2 -= 128

        z = 2 ** 16 * b2 + 2 ** 8 * b1 + b0
        if z < Q:
            return z

        else:
            return None


    def coeff_from_half_byte(self, b):
        if self.eta == 2 and b < 15:
            return 2 - (b % 5)
```

```
        else:
            if self.eta == 4 and b < 9:
                return 4 - b
            else:
                return None


    def simple_bit_pack(self, w, b):
        z = []

        for i in range(256):
            z += self.integer_to_bits(w[i], b.bit_length())

        return self.bits_to_bytes(z)


    def bit_pack(self, w, a, b):
        z = []

        for i in range(256):
            z += self.integer_to_bits(b - w[i], (a + b).bit_length())

        return self.bits_to_bytes(z)


    def simple_bit_unpack(self, v, b):
        c = b.bit_length()
        z = self.bytes_to_bits(v)
        w = [None for _ in range(256)]

        for i in range(256):
            w[i] = self.bits_to_integer(z[i * c : i * c + c], c)

        return w


    def bit_unpack(self, v, a, b):
        c = (a + b).bit_length()
        z = self.bytes_to_bits(v)
        w = [None for _ in range(256)]

        for i in range(256):
            w[i] = b - self.bits_to_integer(z[i * c : i * c + c], c)

        return w


    def hint_bit_pack(self, h):
        y = [0 for _ in range(self.omega + self.k)]
        index = 0

        for i in range(self.k):
            for j in range(256):
                if h[i][j] != 0:
                    y[index] = j
                    index += 1
```

```
            y[self.omega + i] = index

        return y


    def hint_bit_unpack(self, y):
        h = [[0 for _ in range(256)] for _ in range(self.k)]
        index = 0

        for i in range(self.k):
            if y[self.omega + i] < index or y[self.omega + i] > self.omega:
                return None

            while index < y[self.omega + i]:
                h[i][y[index]] = 1
                index += 1

        while index < self.omega:
            if y[index] != 0:
                return None
            index += 1

        return h


    def pk_encode(self, rho, t1):
        pk = self.bits_to_bytes(rho)

        for i in range(self.k):
            pk += self.simple_bit_pack(t1[i], 2 ** ((Q - 1).bit_length() - s

        return pk


    def pk_decode(self, pk):
        y = pk[:32]

        pk_z = pk[32:]
        chunk_size = len(pk_z) // self.k
        z = [pk_z[i : i + chunk_size] for i in range(0, len(pk_z), chunk_siz

        t1 = [None for _ in range(self.k)]

        rho = self.bytes_to_bits(y)

        for i in range(self.k):
            t1[i] = self.simple_bit_unpack(z[i], 2 ** ((Q - 1).bit_length()

        return rho, t1


    def sk_encode(self, rho, K, tr, s1, s2, t0):
        sk = self.bits_to_bytes(rho) + self.bits_to_bytes(K) + self.bits_to_

        for i in range(self.l):
            sk += self.bit_pack(s1[i], self.eta, self.eta)
```

```
        for i in range(self.k):
            sk += self.bit_pack(s2[i], self.eta, self.eta)

        for i in range(self.k):
            sk += self.bit_pack(t0[i], 2 ** (self.d - 1) - 1, 2 ** (self.d -

        return sk


    def sk_decode(self, sk):
        f, g, h = sk[:32], sk[32 : 64], sk[64 : 128]
        sk_y_len = 32 * (2 * self.eta).bit_length() * self.l
        sk_y = sk[128 : 128 + sk_y_len]
        sk_z_len = 32 * (2 * self.eta).bit_length() * self.k
        sk_z = sk[128 + sk_y_len : 128 + sk_y_len + sk_z_len]
        sk_w_len = 32 * self.d * self.k
        sk_w = sk[128 + sk_y_len + sk_z_len : 128 + sk_y_len + sk_z_len + sk

        y = [sk_y[i : i + len(sk_y) // self.l] for i in range(0, len(sk_y),
        z = [sk_z[i : i + len(sk_z) // self.k] for i in range(0, len(sk_z),
        w = [sk_w[i : i + len(sk_w) // self.k] for i in range(0, len(sk_w),

        rho = self.bytes_to_bits(f)
        K = self.bytes_to_bits(g)
        tr = self.bytes_to_bits(h)

        s1 = [None for _ in range(self.l)]
        for i in range(self.l):
            s1[i] = self.bit_unpack(y[i], self.eta, self.eta)

        s2 = [None for _ in range(self.k)]
        for i in range(self.k):
            s2[i] = self.bit_unpack(z[i], self.eta, self.eta)

        t0 = [None for _ in range(self.k)]
        for i in range(self.k):
            t0[i] = self.bit_unpack(w[i], 2 ** (self.d - 1) - 1, 2 ** (self.

        return rho, K, tr, s1, s2, t0


    def sig_encode(self, ct, z, h):
        sigma = self.bits_to_bytes(ct)

        for i in range(self.l):
            sigma += self.bit_pack(z[i], self.gamma1 - 1, self.gamma1)

        sigma += self.hint_bit_pack(h)

        return sigma


    def sig_decode(self, sigma):
        w = sigma[: self.lmbda // 4]
        sigma_x_len = self.l * 32 * (1 + (self.gamma1 - 1).bit_length())
```

```
        sigma_x = sigma[self.lmbda // 4 : self.lmbda // 4 + sigma_x_len]
        sigma_y_len = self.omega + self.k
        sigma_y = sigma[self.lmbda // 4 + sigma_x_len : self.lmbda // 4 + si

        x = [sigma_x[i : i + len(sigma_x) // self.l] for i in range(0, len(s

        ct = self.bytes_to_bits(w)

        z = [None for _ in range(self.l)]
        for i in range(self.l):
            z[i] = self.bit_unpack(x[i], self.gamma1 - 1, self.gamma1)

        h = self.hint_bit_unpack(sigma_y)

        return ct, z, h


    def w1_encode(self, w1):
        w1t = []

        for i in range(self.k):
            w1t += self.bytes_to_bits(self.simple_bit_pack(w1[i], int((Q - 1

        print(w1t)
        return w1t


    def sample_in_ball(self, rho):
        c = [0 for _ in range(256)]
        k = 8

        for i in range(256 - self.tau, 256):
            while self.bits_to_bytes(jth_byte(rho, k, H))[0] > i:
                k += 1

            j = self.bits_to_bytes(jth_byte(rho, k, H))[0]
            c[i] = c[j]
            c[j] = -1 ** H(rho, 8 * (i + self.tau - 256 + 1))[i + self.tau -
            k += 1

        return c


    def rej_ntt_poly(self, rho):
        j = 0
        c = 0
        ac = [None for _ in range(256)]

        while j < 256:
            H_128_c = self.bits_to_bytes(jth_byte(rho, c, H128))[0]
            H_128_c1 = self.bits_to_bytes(jth_byte(rho, c + 1, H128))[0]
            H_128_c2 = self.bits_to_bytes(jth_byte(rho, c + 2, H128))[0]
            ac[j] = self.coeff_from_three_bytes(H_128_c, H_128_c1, H_128_c2)

            c += 3
            if ac[j] != None:
```

```
            j += 1

        return ac


    def rej_bounded_poly(self, rho):
        j = 0
        c = 0
        a = [None for _ in range(256)]

        while j < 256:
            z = self.bits_to_bytes(jth_byte(rho, c, H))[0]
            z0 = self.coeff_from_half_byte(z % 16)
            z1 = self.coeff_from_half_byte(z // 16)

            if z0 != None:
                a[j] = z0
                j += 1

            if z1 != None and j < 256:
                a[j] = z1
                j += 1

            c += 1

        return a


    def expand_a(self, rho):
        Ac = [[None for _ in range(self.l)] for _ in range(self.k)]

        for r in range(self.k):
            for s in range(self.l):
                Ac[r][s] = self.rej_ntt_poly(rho + self.integer_to_bits(s, 8

        return Ac


    def expand_s(self, rho):
        s1 = [None for _ in range(self.l)]
        s2 = [None for _ in range(self.k)]

        for r in range(self.l):
            s1[r] = self.rej_bounded_poly(rho + self.integer_to_bits(r, 16))

        for r in range(self.k):
            s2[r] = self.rej_bounded_poly(rho + self.integer_to_bits(r + sel

        return s1, s2


    def expand_mask(self, rho, mu):
        c = 1 + (self.gamma1 - 1).bit_length()
        s = [None for _ in range(self.l)]

        for r in range(self.l):
```

```
            n = self.integer_to_bits(mu + r, 16)
            v = [self.bits_to_bytes(jth_byte(rho + n, 32 * r * c + i, H))[0]
            s[r] = self.bit_unpack(v, self.gamma1 - 1, self.gamma1)

        return s


    def power_2_round(self, r):
        rp = r % Q
        r0 = mod_plus_minus(rp, 2 ** self.d)

        return int((rp - r0) / 2 ** self.d), r0


    def decompose(self, r):
        rp = r % Q
        r0 = mod_plus_minus(rp, 2 * self.gamma2)

        if rp - r0 == Q - 1:
            r1 = 0
            r0 -= 1

        else:
            r1 = (rp - r0) / (2 * self.gamma2)

        return int(r1), int(r0)


    def high_bits(self, r):
        r1, r0 = self.decompose(r)
        return r1


    def low_bits(self, r):
        r1, r0 = self.decompose(r)
        return r0


    def make_hint(self, z, r):
        r1 = self.high_bits(r)
        v1 = self.high_bits(r + z)

        return r1 != v1


    def use_hint(self, h, r):
        m = (Q - 1) // (2 * self.gamma2)
        r1, r0 = self.decompose(r)

        if h == 1 and r0 > 0:
            return (r1 + 1) % m

        if h == 1 and r0 <= 0:
            return (r1 - 1) % m

        return r1
```

```
    def ntt(self, w):
        wc = [None for _ in range(256)]
        for j in range(256):
            wc[j] = w[j]

        k = 0
        len = 128

        while len >= 1:
            start = 0
            while start < 256:
                k += 1
                zeta = pow(self.zeta, brv(k), Q)
                for j in range(start, start + len):
                    t = (zeta * wc[j + len]) % Q
                    wc[j + len] = (wc[j] - t) % Q
                    wc[j] = (wc[j] + t) % Q

                start += 2 * len

            len //= 2

        return wc


    def ntt_inv(self, wc):
        w = [None for _ in range(256)]
        for j in range(256):
            w[j] = wc[j]

        k = 256
        len = 1

        while len < 256:
            start = 0
            while start < 256:
                k -= 1
                zeta = -pow(self.zeta, brv(k), Q)

                for j in range(start, start + len):
                    t = w[j]
                    w[j] = (t + w[j + len]) % Q
                    w[j + len] = (t - w[j + len]) % Q
                    w[j + len] = (w[j + len] * zeta) % Q

                start += 2 * len

            len *= 2

        f = 8347681
        for j in range(256):
            w[j] = (w[j] * f) % Q

        return w
```

Testes de aplicação

Para efeitos de teste, desenvolveu-se a função **ml_dsa_test**, responsável por, através dos respetivos parâmetros, dependendo do conjunto pretendido, gerar as respetivas chaves, pública e privada (a função de teste não testa as funções de assinatura e verificação, uma vez que não se encontram definidas).

In [28]:
```
def ml_dsa_test(algorithm):
    if algorithm == 'ML-DSA-44':
        mldsa = MLDSA(39, 128, 2 ** 17, 88, 4, 4, 2, 80)

    elif algorithm == 'ML-DSA-65':
        mldsa = MLDSA(49, 192, 2 ** 19, 32, 6, 5, 4, 55)

    elif algorithm == 'ML-DSA-87':
        mldsa = MLDSA(60, 256, 2 ** 19, 32, 8, 7, 2, 75)

    else:
        print('[ML-DSA] invalid parameter set')
        return


    pk, sk = mldsa.ml_dsa_keygen()
    print(f'[ML-DSA] ({algorithm}) keys generated with success')
```

ML-DSA-44

In [29]:
```
ml_dsa_test('ML-DSA-44')
```

```
[ML-DSA] (ML-DSA-44) keys generated with success
```

ML-DSA-65

In [30]:
```
ml_dsa_test('ML-DSA-65')
```

```
[ML-DSA] (ML-DSA-65) keys generated with success
```

ML-DSA-87

In [31]:
```
ml_dsa_test('ML-DSA-87')
```

```
[ML-DSA] (ML-DSA-87) keys generated with success
```

Invalid parameter set

In [32]:
```
ml_dsa_test('ML-DSA-LIONEL-MESSI')
```

```
[ML-DSA] invalid parameter set
```