Dilithium

Neste notebook implementamos o algoritmo *Dilithium*, um esquema de assinatura digital candidato ao concurso NIST-PQC.

Como implementação, fornecemos uma classe instanciável onde a geração das chaves é feita no construtor e a assinatura e verificação são fornecidos como métodos.

In [1]:
```
from cryptography.hazmat.primitives import hashes
```

Parametros

Como este algoritmo tem como um dos objetivos ser modular e parameterizavel, fornecemos vários modos de instancia para o *dilithium* com níveis de segurança nos parametros, de seguida encontram-se as classes que definem estes parametros para cada nivel de segurança, estas classes são passadas como argumento ao construtor do *Dilithium*.

In [2]:
```
class Weak:
    k = 3
    l = 2
    eta = 7
    beta = 375
    omega = 64

class Medium:
    k = 4
    l = 3
    eta = 6
    beta = 325
    omega = 80

class Recommended:
    k = 5
    l = 4
    eta = 5
    beta = 275
    omega = 96

class VeryHigh:
    k = 6
    l = 5
    eta = 3
    beta = 175
    omega = 120
```

Implementação

De seguida encontra-se a implementação realizada pelo grupo. Como jupyter notebook não permite inserir blocos de markdown entre metodos da classe, no código a seguir encontram-se alguns comentários relevantes na implementação e notas informativas, alternativamente uma descrição do procedimento encontra-se no bloco de markdown a seguir

In [3]:
```
class Dilithium:
    def __init__(self, params=Recommended):
        # Define Parameters
        self.n = 256
        self.q = 8380417
        self.d = 14
        self.weight = 60
        self.gamma1 = 523776 #(self.q-1) / 16
        self.gamma2 = 261888 #self.gamma1 / 2
        self.k = params.k
        self.l = params.l
        self.eta = params.eta
        self.beta = params.beta
        self.omega = params.omega

        # Define Fields
        Zq.<x> = GF(self.q)[]
        self.Rq = Zq.quotient(x^self.n+1)

        # Generate Keys
        self.A = self.expandA()
        self.s1 = self.sample(self.eta, self.l)
        self.s2 = self.sample(self.eta, self.k)
        self.t = self.A * self.s1 + self.s2
        # Public Key : A, t
        # Private Key : s1, s2

    def sign(self, m):
        z = None
        while z == None:
            y = self.sample(self.gamma1-1, self.l)
            # Ay é reutilizado por isso precalcula-se
            Ay = self.A * y
            w1 = self.high_bits(self.A * y, 2 * self.gamma2)
            c = self.H(b"".join([bytes([ int(i) for i in e ]) for e in w1]) + m)
            c_poly = self.Rq(c)
            z = y + c_poly * self.s1

            if (self.sup_norm(z) >= self.gamma1 - self.beta) and (self.sup_norm([self
                z = None
        return (z,c)


    def verify(self, m, sig):
        (z,c) = sig
        w1_ = self.high_bits(self.A*z - self.Rq(c)*self.t, 2*self.gamma2)
        torf1 = (self.sup_norm(z) < self.gamma1-self.beta)
        torf2 = (c == self.H(b"".join([bytes([ int(i) for i in e ]) for e in w1_]) + m
        return torf1 and torf2

    ########## Auxiliar Functions ##########

    # The function ExpandA maps a uniform seed  ∈ {0, 1}^256 to a matrix A ∈ Rq^k×l
    def expandA(self):
        # Na submissão original assume-se ρ como uma
        # seed uniforme para amostrar aleatoriamente
        # neste caso considera-se que `random_element`
        # tem o valor equivalente da seed internamente
        mat = [ self.Rq.random_element() for _ in range(self.k*self.l) ]
        return matrix(self.Rq, self.k, self.l, mat)

    def sample(self, coef_max, size):
        def rand_poly():
            return self.Rq([randint(0,coef_max) for _ in range(self.n)])

        vector = [ rand_poly() for _ in range(size) ]

        # Vectores sao representados sob
        # a forma de matrizes para permitir as
        # operações com a matriz A
        return matrix(self.Rq,size,1,vector)

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

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

    def decompose(self, r, alfa):
        # Nota: Na submissão original é assumido
        # que as operações no decompose são aplicadas
        # a cada coeficiente.
        #     r1 r0
        r0_vector = []
        r1_vector = []
        torf = True
        for p in r:
            r0_poly = []
            r1_poly = []
            for c in p[0]:
                c = int(mod(c,int(self.q)))
                r0 = int(mod(c,int(alfa)))
                if c - r0 == int(self.q) - int(1):
                    r1 = 0
                    r0 = r0 - 1
                else:
                    r1 = (c - r0) / int(alfa)
                r0_poly.append(r0)
                r1_poly.append(r1)
            if torf:
                #print("AAAAAAAAAAAAAAAA",self.Rq(r0_poly))
                torf = False
            r0_vector.append(self.Rq(r0_poly))
            r1_vector.append(self.Rq(r1_poly))
        # Como já não vamos realizar mais operações
        # sobre matrizes então podemos apenas utilizar
        # listas de python para estes vectors
        return (r1_vector, r0_vector)

    def H(self, obj):
        sha3 = hashes.Hash(hashes.SHAKE256(int(60)))
        sha3.update(obj)
        res = [ (-1) ** (b % 2) for b in sha3.finalize() ]
        return res + [0]*196

    # https://en.wikipedia.org/wiki/Uniform_norm
    def sup_norm(self, v):
        return max([ max(p[0]) for p in v])
```

Descrição

O documento de referencia pode ser consultado em: https://eprint.iacr.org/2017/633.pdf

Geração das chaves

O algoritmo de geração de chaves gera uma matriz A de dimensões k×l, e amostra 2 vetores s1 e s2 de tamanhos l×1 e k×1 respetivamente, e ainda gera o ultimo parametro publico t = A*s1 + s2.

Para amostrar a matriz A e os vetores de polinómios s1 e s2 fizemos 2 métodos auxiliares (`expandA` e `sample`)

Depois de gerar estes valores todos temos finalmente as chaves publica e privada

Public Key: (A, t)
Private Key: (A, t, s1, s2)

Assinatura

A assinatura obtem-se com os seguintes passos:

- É amostrado *y* com dimenção l×1 e de seguida calcula-se os high_bits de *A*y para* w1*
- Obter o hash `H()` a partir de *w1* e da mensagem
- Calcular `z = y + c*s1`
- Verificar a condição de assinatura, caso contrário, voltar ao inicio

Verificação

Para se verificar a assinatura a partir da chave pública, têm-se os seguintes passos:

- Calcula-se os high_bits de `A * y - c * t` para *w1*
- Confirma-se a condição da assinatura se verifica

> **Nota:** Todas as funções auxiliares utilizadas no âmbito do algoritmo, estão defenidas na classe

Testes

Test 1

Verificar se o esquema valida corretamente uma assinatura.

In [4]:
```
dilithium = Dilithium(params=Weak)
sig = dilithium.sign(b"ola mundo cruel")
print("Test 1 (Must be True):",dilithium.verify(b"ola mundo cruel", sig))
```

```
Test 1 (Must be True): True
```

Test 2

Verificar se o esquema reconhece quando os dados assinados são diferentes

In [5]:
```
sig = dilithium.sign(b"ola mundo cruel")
print("Test 2 (Must be False):",dilithium.verify(b"adeus mundo cruel", sig))
```

```
Test 2 (Must be False): False
```

Test 3

Verificar se entre instancias diferentes não há relações

In [6]:
```
dilithium_other = Dilithium(params=Weak)
sig = dilithium.sign(b"ola mundo cruel")
print("Test 3 (Must be False):",dilithium_other.verify(b"ola mundo cruel",sig))
```

```
Test 3 (Must be False): False
```

In [ ]:
```

```