qTesla

Neste notebook implementamos o algoritmo *qTesla*, 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]:

```
import os
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 *qTesla* com níveis de segurança nos parametros (`p-I` e `p-III`), de seguida encontram-se as classes que definem estes parametros, estas classes são passadas como argumento ao construtor do *qTesla*.

In [2]:

```
class pI:
    n = 1024
    sigma = 8.5
    q = 343576577
    h = 25
    Le = 554
    Ls = 554
    B = 2^19 - 1
    d = 22
    k = 4

class pIII:
    n = 2048
    sigma = 8.5
    q = 856145921
    h = 40
    Le = 901
    Ls = 901
    B = 2^21 - 1
    d = 24
    k = 5
```

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 qTesla:
    def __init__(self, params=pI):
        # Define Parameters
        self.n = params.n
        self.sigma = params.sigma
        self.q = params.q
        self.h = params.h
        self.Le = params.Le
        self.Ls = params.Ls
        self.E = self.Le
        self.S = self.Ls
        self.B = params.B
        self.d = params.d
        self.k = params.k
        self.K = 256
        self.alfa = self.sigma / self.q

        # Define sup_norm()
        self.sup_norm = max

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

        # Generate Keys
        counter = 1
        pre_seed = os.urandom(self.K // 8)
        seed_s, seeds_e, seed_a, seed_y = self.prf1(pre_seed)
        a = self.genA(seed_a)

        while True:
            s = self.GaussSampler(seed_s, counter)
            counter += 1
            if self.checkS(s):
                break

        e = []
        t = []
        for i in range(self.k):
            while True:
                e_i = self.GaussSampler(seeds_e[i], counter)
                counter += 1
                if self.checkE(e_i):
                    e.append(e_i)
                    break
            t.append(a[i] * s + e[i])
            #t.append(self.mod_list(self.poly_add(self.poly_mul(a[i], s) , e[i]) , se

        g = self.G(t)

        # Public Key
        self.pub_key = (t, seed_a)
        # Private Key
        self.priv_key = (s, e, seed_a, seed_y, g)

    def sign(self, m):
        s, e, seed_a, seed_y, g = self.priv_key

        counter = 1
        r = os.urandom(self.K // 8)
        rand = self.prf2(seed_y, r, self.G(m))

        while True:
            y = self.ySampler(rand, counter)
            a = self.genA(seed_a)

            v = []
            for i in range(self.k):
                # a[i] * y
                v.append(self.mod_list(self.poly_mul(a[i], y), self.q))

            self.tmp = v
            # print("v:",v[0][0])
            c_prime = self.H(v, self.G(m), g)
            c = self.sparse_to_list(self.Enc(c_prime))
            #print(len(list(filter(lambda x: x < 0, c))))
            #sc = self.sparse_mul(self.Enc(c_prime), s)
            #print("sc",sc)
            #print("Rq(sc)",self.Rq(sc))

            #z = y + s*c

            #print(len(list(filter(lambda x: x < 0, self.poly_mul(s, c)))))

            z = self.poly_add(y, self.poly_mul(s, c))
            # print(len(list(filter(lambda x: x < 0, z))))

            # Check if belongs to R[B-S]
            belongs = True
            tmp = abs(self.B - self.S)
            for coef in z:
                #print("c:",c)
                #print("B-S:",tmp)
                if abs(coef) > tmp:
                    belongs = False

            # if not belongs:
            #     counter += 1
            #     continue

            w = []
            for i in range(self.k):
                w.append(self.mod_list(self.poly_sub(v[i], self.poly_mul(e[i], c)), se
                if self.sup_norm(w[i]) >= 2**(self.d-1) - self.E or self.sup_norm(w[i
                    counter += 1
                    continue
                    #torf = True
                    #break
            #if not torf:
            return (z, c_prime)

    def verify(self, m, sig):
        t, seed_a = self.pub_key
        z, c_prime = sig
        c = self.sparse_to_list(self.Enc(c_prime))
        a = self.genA(seed_a)

        w = []
        for i in range(self.k):
            w.append(self.mod_list(self.poly_sub(self.poly_mul(a[i], z), self.poly_mu
            #self.poly_sub(self.poly_mul(a[i], z), self.poly_mul(t[i], c))
            # w.append(a[i] * z - t[i]*c)

        # Check if belongs to R[B-S]
        belongs = True
        for c in z:
            if c > abs(self.B - self.S):
                belongs = False

        #print("val1:",c_prime)
        # print("v:",w[0])

        # for c1, c2 in zip(w[0], self.tmp[0]):
        #     if c1 != c2:
        #         print("Different:", c1, c2)
        # print("v:",w[0] == self.tmp[0])
        if c_prime != self.H(w,self.G(m),self.G(t)) : # or not belongs
            return False
        return True

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

    def checkE(self, e):
        res = 0
        e_list = list(e)
        e_list.sort(reverse=True)
        for i in range(0,self.h):
            res += e_list[i]

        return (res > self.Le)

    def checkS(self, s):
        res = 0
        s_list = list(s)
        s_list.sort(reverse=True)
        for i in range(0,self.h):
            res += s_list[i]

        return (res > self.Ls)


    def prf1(self, pre_seed):
        elem = 3 + self.k
        xof = hashes.Hash(hashes.SHAKE256(int(self.K*(self.k + 3)/8)))
        xof.update(pre_seed)
        seed = xof.finalize()

        seed_s = seed[0:32]
        seeds_e = [ seed[32*i:32*(i+1)] for i in range(1,self.k+1)]
        seed_a = seed[self.k+1:self.k+2]
        seed_y = seed[self.k+2:self.k+3]

        return seed_s, seeds_e, seed_a, seed_y

    def prf2(self, seed, r, g_m):
        xof = hashes.Hash(hashes.SHAKE256(int(self.K // 8)))
        xof.update(seed)
        xof.update(r)
        xof.update(g_m)
        return xof.finalize()

    def genA(self, seed_a):
        # Convert seed_a to int and set seed for
        # sagemath's random generator
        set_random_seed(int.from_bytes(seed_a, "big"))
        return [ self.Rq.random_element() for _ in range(self.k) ]

    def GaussSampler(self, seed_s, nounce):
        seed_nounce = int.from_bytes(seed_s, "big") + nounce
        set_random_seed(seed_nounce)

        # If the distribution 'gaussian' is specified,
        # the output is sampled from a discrete Gaussian
        # distribution with parameter σ=x
        #return self.R.random_element(x=self.sigma,distribution='gaussian')
        return self.Rq.random_element(x=self.sigma,distribution='gaussian')

    def G(self, m):
        if type(m) == list:
            # Convert poly to bytes form
            m = b''.join([ b''.join([ int(c).to_bytes(4,"big") for c in p ]) for p in

        xof = hashes.Hash(hashes.SHAKE256(int(40)))
        xof.update(m)
        return xof.finalize()

    # FIXME: Provavelmente mal defenido
    def ySampler(self, seed, nounce):
        seed_nounce = int.from_bytes(seed, "big") + nounce
        set_random_seed(seed_nounce)
        return self.R.random_element(x=-self.B, y=self.B+1, distribution='uniform')

    def H(self, v, g_m, g_t):
        pow_2_d = 2**self.d
        w = []
        for i in range(self.k):
            for j in range(self.n):
                val = int(v[i][j]) % (pow_2_d)

                if val > 2**(self.d-1):
                    val -= pow_2_d
                wij = (v[i][j] - val) // pow_2_d
                w.append(int(wij).to_bytes(1,"big"))

        w = b''.join(w + [g_m, g_t])
        xof = hashes.Hash(hashes.SHAKE256(int(self.K // 8)))
        xof.update(w)
        return xof.finalize()

    def Enc(self, c_prime):
        D = 0
        cnt = 0
        rate_xof = 168
        r = self.cSHAKE128(c_prime, rate_xof, D)

        pos_list = []
        sign_list = []

        i = 0
        c = [0] * self.n
        while i < self.h:
            if(cnt > (rate_xof - 3)):
                D += 1
                cnt = 0
                r = self.cSHAKE128(c_prime, rate_xof, D)
            pos = int.from_bytes(r[cnt:(cnt+2)],'big') % self.n
            if c[pos] == 0:
                c[pos] = -1 if (r[cnt+2] % 2 == 1) else 1
                pos_list.append(pos)
                sign_list.append(c[pos])
                i += 1
            cnt += 3
        return (pos_list, sign_list)

    def cSHAKE128(self, c_prime, rate, D):
        xof = hashes.Hash(hashes.SHAKE256(int(rate)))
        xof.update(int(D).to_bytes(136,"big") + c_prime)
        return xof.finalize()

    def sparse_to_list(self, c):
        pos_list, sign_list = c
        poly_list = [0] * self.n
        for pos, sign in zip(pos_list,sign_list):
            poly_list[pos] = sign
        return poly_list

    # def sparse_mul(self, c, poly):
    #     (pos_list,sign_list) = c
    #     f = [0] * self.n
    #     for i in range(0,self.h):
    #         pos = pos_list[i]
    #         for j in range(0,pos):
    #             f[j] = f[j] - sign_list[i]*poly[j+self.n-pos]
    #         for j in range(pos,self.n):
    #             f[j] = f[j] + sign_list[i]*poly[j-pos]
    #     return f

    def poly_mul(self, p1, p2):
        return [ int(c1)*int(c2) for c1, c2 in zip(p1, p2) ]

    def poly_add(self, p1, p2):
        return [ int(c1)+int(c2) for c1, c2 in zip(p1, p2) ]

    def poly_sub(self, p1, p2):
        return [ int(c1)-int(c2) for c1, c2 in zip(p1, p2) ]

    def mod_list(self, l, m):
        return [ mod(c,m) for c in l ]
```

Descrição

O documento de referencia pode ser consultado em: https://eprint.iacr.org/2019/085.pdf

Geração das chaves

A geração de chaves parte da geração das *seeds* que são usadas para a amostragem das váriáveis necessárias a partir da `prf1`. Depois gera-se *k* elementos de Rq uniformemente para a variável *a*. De seguida gera-se *s* a partir de uma distribuição gaussiana discreta centrada com desvio padrão *sigma*, até que `checkS(s)` se verifique. Finalmente gera-se da mesma forma *k* polinómios para *e* até que `checkE(e)` se verifique e ainda se compões em *t*, *k* polinómios que resultam de `t_i = a_i * s * e_i mod q`.

Depois de gerar estes valores todos temos finalmente as chaves publica e privada depois de se ter o *digest* de *t* (que é chamado de *g*)

Public Key: (t, seed_a)
Private Key: (s, e, seed_a, seed_y, g)

Assinatura

A assinatura obtem-se com os seguintes passos:

- Amostrar *y* com `ySample` em R[B], *rand*).
- Calcular o hash para ter o *c_prime* a partir de `H()` com *m*
- *c* é gerado sob a forma de um polinómio esparso (posições e valores) a partir de *c_prime*.
- Obtem-se `z = y + s*c`
- Verifica-se se *z* não está em R[B−S]
- Verifica-se *correctess* para *k* polinómios em que `w = v-e*c mod q`
- Obtem-se a assinatura `(z, c_prime)`

Verificação

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

- *c* é gerado sob a forma de um polinómio esparso (posições e valores) a partir de *c_prime*.
- Calcula-se *w* em *k* polinómios a partir de `w = a * z - t * c mod q`
- 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]:

```
qtesla = qTesla(params=pI)
sig = qtesla.sign(b"ola mundo cruel")
result = qtesla.verify(b"ola mundo cruel",sig)
print("Test 1 (Must be True):", result)
```

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

Test 2

Verificar se o esquema reconhece quando os dados assinados são diferentes

In [5]:

```
sig = qtesla.sign(b"ola mundo cruel")
print("Test 2 (Must be False):", qtesla.verify(b"adeus mundo cruel", sig))
```

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

Test 3

Verificar se entre instancias diferentes não há relações

In [6]:

```
qtesla_other = qTesla(params=pI)
sig = qtesla.sign(b"ola mundo cruel")
print("Test 3 (Must be False):", qtesla_other.verify(b"ola mundo cruel",sig))
```

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