qTESLA

June 23, 2020

1 Trabalho Prático 4

1.1 qTESLA

In [1]:
```
import hashlib

# Parâmetros

# K = 256
k = 5 # number of public polynomials
h = 40 # number of nonzero entries of output elements of Enc
Les = 901   # bound in checkE and checkS
B = 2^21-1   # determines interval randomness during signing
d = 24   # number of rounded bit
n = 2048 # dimension
q = 856145921 # modulus


## Os anéis usuais com o módulo ciclotómico

R_.<w> = ZZ[]
R.<x>  = QuotientRing(R_,R_.ideal(w^n+1))

Rq_.<w> = GF(q)[]
Rq.<x>  = QuotientRing(Rq_,Rq_.ideal(w^n+1))
```

1.2 Funções auxiliares

In [2]:
```
class NTT(object):
#
    def __init__(self, n):
        if not any([n == t for t in [32,64,128,256,512,1024,2048]]):
            raise ValueError("improper argument ",n)
        self.n = n
        self.F = GF(q) ;
        self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen();
```

```
        self.w = w

        g = (w^n + 1)
        xi = g.roots(multiplicities=False)[-1]
        self.xi = xi
        rs = [xi^(2*i+1)  for i in range(n)]
        self.base = crt_basis([(w - r) for r in rs])


    def ntt(self,f):
        def _expand_(f):
            u = f.list()
            return u + [0]*(self.n-len(u))

        def _ntt_(xi,N,f):
            if N==1:
                return f
            N_ = N/2
            xi2 =  xi^2
            f0 = [f[2*i]   for i in range(N_)]
            f1 = [f[2*i+1] for i in range(N_)]
            ff0 = _ntt_(xi2,N_,f0)
            ff1 = _ntt_(xi2,N_,f1)

            s  = xi
            ff = [self.F(0) for i in range(N)]

            for i in range(N_):
                a = ff0[i]
                b = s*ff1[i]
                ff[i] = a + b
                ff[i + N_] = a - b
                s = s * xi2
            return ff

        return _ntt_(self.xi,self.n,_expand_(f))

    def ntt_inv(self,ff):                              ## transformada inversa
        return sum([ff[i]*self.base[i] for i in range(self.n)])

# Instancia NTT
ntt = NTT(n)
```

In [3]:
```
def enc(output):

    count = 0
    positionList = [0] * h
    signList = [0] * h
```

```
        # Use Rejection Sampling to Determine Positions to be Set in the New Vector

        C = [0] * n
        random_enc = G(output, 168)

        for i  in range(h):

            if count > 168 - 3:

                count = 0


            position = (random_enc[count] << 8) | (random_enc[count + 1] & 0xFF)
            position &= (n - 1)

            # Position is between [0, n - 1] and Has not Been Set Yet

            if C[position] == 0:

                if (random_enc[count + 2] & 1) == 1:

                    C[position] = -1

                else:

                    C[position] = 1


                positionList[i] = position
                signList[i] = C[position]
                i += 1


            count += 3

        return positionList, signList
```

```
In [4]: def multi_SC(s, c):

        pos_list, sign_list = c
        prod = [0] * n

        for i in range(h):

            pos = pos_list[i]

            for j in range(pos):
```

```
                prod[j] = prod[j] - sign_list[i] * s[j + n - pos]

            for j in range(pos, n):

                prod[j] = prod[j] + sign_list[i] * s[j - pos]
        prod = Rq(prod)

        return prod
```

```
In [5]: def multi_pol(a, b):

            a_ntt = ntt.ntt(a)
            b_ntt = ntt.ntt(b)
            a_b = [num1 * num2 for num1, num2 in zip(a_ntt, b_ntt)]

            return ntt.ntt_inv(a_b)
```

```
In [6]: def G(s, l=32):

            hashing = hashlib.shake_256()
            hashing.update(s)

            return hashing.digest(int(l))


        def checkES(s):
            soma = 0
            s_list = list(s)
            s_list.sort(reverse=True)
            for i in range(h):
                soma += s_list[i]
            if soma > Les:
                return 1
            return 0


        def GenA():
            return Rq.random_element()


        def H(v, hash_m):
            w = [0] * n * k
            for i in range(k):
                v_list = list(v[i])

                for j in range(n):
```

```
                val = mod(v_list[j], 2^d)
                if val > 2^(d-1):
                    val = val - 2^d
                w[j] = (int(v_list[j]) - int(val)) // 2^d

        return G(bytes(str(w), 'utf-8') + hash_m)
```

In [7]:
```
def keygen():

    # Gera os polinômios a  Rq
    a = [GenA() for _ in range(k)]

    # Gera o polinômio s  R através da distribuição gaussiana ( = 8.5)
    # Verifica se a soma das h maiores entradas for superior a  Ls
    while True:
        s = R.random_element(x=8.5, distribution="gaussian")
        if checkES(s) == 0:
            break

    # Gera k polinômios e  R através da distribuição gaussiana ( = 8.5)
    # Verifica se a soma das h maiores entradas é superior a Le
    e = []
    for i in range(k):

        while True:
            e_item = R.random_element(x=8.5, distribution="gaussian")
            if checkES(e_item) == 0:
                e.append(e_item)
                break

    # Calcula t_i = a_i * s + e_i para i de 0 a k - 1
    a_s = [multi_pol(a[i], s) for i in range(k)]

    t = a_s + e

    sk = s, e, a
    pk = a, t

    return sk, pk


def sign(m, sk):

    s, e, a = sk

    # Gera uniformimente o polinômio y com coeficientes entre -B e B
    y = Rq.random_element(x=-B, y=B+1, distribution="uniform")
```

```
    # Calcula v_i = a_i * y para i de 0 a k - 1
    v = [multi_pol(a[i], y) for i in range(k)]

    c_ = H(v, G(m))

    c = enc(c_)

    # Calcula z = y + s * c
    s_c = multi_SC(s, c)

    z = y + s_c

    return (z, c_)


def verify(m, sig, pk):

    z, c_ = sig
    a, t = pk

    c = enc(c_)

    # Calcula w_i = a_i * z - t_i * c  para i de 0 a k - 1
#     t_ntt = ntt.ntt(t)
    a_z = [multi_pol(a[i], z) for i in range(k)]
    t_c = [multi_SC(t[i], c) for i in range(k)]

    w = [num1 - num2 for num1, num2, in zip(a_z, t_c)]

    if c_ != H(w, G(m)):

        return False

    return True
```

1.3 Teste

In [8]:
```
sk, pk = keygen()

# Mensagem a ser assinada
m = os.urandom(32)

print("Mensagem: ", m)

signature = sign(m, sk)

print("Assinatura correta? ", verify(m, signature, pk))
```

```
Mensagem:  b'\xa6\xdfV0\x93\xee\xe5\xb8\xb7\x86\xa1\x10\xa7\xe2\xa3"\xb5\x84\x89\xf4\x8c\xb3Z\
Assinatura correta?  True
```