# qTESLA

June 5, 2019

## 1 qTESLA

O qTESLA é da família dos esquemas de assinaturas pos-quânticas, que se baseia no problema de decisão do Ring Leaning With Errors (R-LWE). Das diferentes implementações existentes optou-se pela **qTESLA-I** que se baseia numa geração heurística de parâmetros.

```
In [ ]: import numpy as np
        import hashlib
        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import hashes
```

### 1.1 Parâmetros utilizados

- `n`: dimensão
- `k`: número de amostras
- `q`: módulo
- `h`: número de entradas diferentes de 0 nos elementos retornados pela função `Enc`
- `K`: comprimento do output da função `H` e do input da `GenA` e `Enc`.
- `Le`: limite de `checkE`
- `Ls`: limite de `checkS`
- `B`: determina o intervalo de aleatoriedade durante a assinatura.
- `d`: número de bits aleatórios

```
In [ ]: K = 256
        k = 1
        h = 30
        Le = 1586
        Ls = 1586
        B = 2^20-1
        d = 21
        n = 512
        q = 4205569

        Zx.<x>  = ZZ[]
        Gq.<z>  = GF(q)[]

        R.<x> = Zx.quotient(x^n+1)  # R
        Rq.<z> = Gq.quotient(z^n+1) # R/q
```

## 1.2 Funções auxiliares

In [ ]:
```
def hash(s):
    h = hashlib.sha256()
    h.update(s)
    return h.digest()

def binary(size=n):
    return list(np.random.choice([0,1],size))

def GenA():
    return Rq.random_element()

def _center_lift(x):
    return lift(x + q//2) - q//2

def _round(w):
    return Zx(map(lambda x: _center_lift(x), w.list()))

def checkS(s):
    sum = 0
    ls = list(s)
    ls.sort(reverse=True)
    for i in range(0, h):
        sum += ls[i]
    if sum > Ls:
        return 1
    return 0

def checkE(s):
    sum = 0
    ls = list(s)
    ls.sort(reverse=True)
    for i in range(0, h):
        sum += ls[i]
    if sum > Le:
        return 1
    return 0

def H(v, hash_m):
    w = [0] * n
    for i in range(n):
        val = v[i] % 2^d
        if val > 2^(d-1):
            val = val - 2^d
        w[i] =  (v[i] - val)/2^d
    return hash(str(w)+hash_m)
```

## 1.3 qTESLA

O qTESLA está dividido em 3 funções principais: `setup`, `sign` e `verify`.

A função `setup` é responsável por gerar a chave privada (sk) e pública (pk).

1. Gera-se `a` através do anél Rq
2. Escolhe-se $s \in R$ com as entradas geradas através de uma distribuição gaussiana.
3. Repetir o passo 2 enquanto a soma das h maiores entradas for superior a $L_S$.
4. Escolher um $e \in R$ com as entradas geradas através de uma distribuição gaussiana.
5. Repetir o passo 4 enquanto a soma das h maiores entradas for superior a $L_E$.
6. Calcular $t = a * s + e \in R$.
7. Retornar `sk = (a, e, a)` e `pk = (a, t)`.

---

A função `sign` é responsável por assinar uma mensagem `m` através da chave privada `sk`.

1. Gerar um `y` uniformemente de forma aleatória com coeficientes curtos (B-short) em Rq.
2. Multiplicar a componente `a` de `sk` e multiplicar por `y`, ou seja, `v = a*y`
3. Centrar `v` entre [-B,B].
4. c = H(v, m)
5. z = y + s*c
6. Se z não é (B-s)-short então voltar ao passo 1.
7. Se a$y$ - $e$c não está bem arredondado então voltar ao passo 1.
8. Retornar (z, c)

---

A função `verify` é responsável por verificar a assinatura da mensagem através da chave pública `pk`.

1. Rejeitar se `z` não for (B-S)-short.
2. $w = a * z - t * c \in Rq$.
3. Centrar `w` entre [-B,B].
4. Se `c != H(w, m)`, então a assinatura não é válida.
5. A assinatura é válida.

```
In [ ]: def setup():
            a = GenA()
            s = None
            while (True):
                s = Rq.random_element(distribution="gaussian")
                if checkS(s) == 0:
                    break

            e = None
            while (True):
                e = Rq.random_element(distribution="gaussian")
                if checkE(e) == 0:
```

```
            break

    t = a*s + e
    sk = (s, e, a)
    pk = (a, t)
    return sk, pk

def sign(m, sk):
    s, e, a = sk
    y = Rq.random_element(x=-B, y=B+1, distribution="uniform")
    v = _round(a*y)
    c1 = H(v, hash(str(m)))
    c2 = Enc(c1)
    z = y + s*c2
    return (z, c1)

def verify(mess, s, pk):
    z, c1 = s
    c2 = Enc(c1)
    a, t = pk
    w = _round(a*z - t*c2)
    if c1 != H(w, hash(str(mess))):
        return False
    return True


mess = binary(n)
sk, pk = setup()
s = sign(mess,sk)
res = verify(mess, s, pk)

print res
```