qTESLA

In [1]:
```
import hashlib
import random
import math
```

Parâmetros e Anéis

In [2]:
```
K = 256
k = 1 # sample
n = 512 # dimensão
q = 4205569 # módulo
h = 30
L_E = 1586 # limite de E
L_S = 1586 # limite de S
B = (2**20)-1 # intervalo de aleatoriedade na assinatura
d = 21 # #bits aleatórios
```

In [3]:
```
Zx.<x> = ZZ[]
R.<x> = Zx.quotient(x^n + 1)
print(R)

Fq = GF(q)
Fqy.<y> = Fq[]
Rq.<y> = Fqy.quotient(y^n + 1)
print(Rq)
```

```
Univariate Quotient Polynomial Ring in x over Integer Ring with mo
dulus x^512 + 1
Univariate Quotient Polynomial Ring in y over Finite Field of size
4205569 with modulus y^512 + 1
```

KeyGen

Funções Auxiliares

De forma a podermos implementar uma versão da nossa geração de chaves foi necessário implementar-mos alguns algoritmos auxiliares que suportariam a função principal.

Assim nesta célula encontraremos algoritmos que geram polinómios (**a,e,s**) e ainda algoritmos que nos indicam a validade de um dado polinómio.

In [4]:
```
'''
Gen_A() : Devolve um polinómio aleatório em Rq.
Este é gerado através da função pré-definida $$ random_element() $$
'''
```

```python
def Gen_A():
    return Rq.random_element()


'''
generate_E(): Devolve um polinómio aleatório em Rq, com distribuiçã
Este é gerado através da função pré-definida $$ random_element() $$
Ao polinómio gerado é aplicado um teste de limites através da funçã
'''
def generate_E():
    e = None
    while True:
        e = Rq.random_element(distribution="gaussian")
        if checkE(e) == 0:
            return e


'''
checkE(e): Recebe um polinómio em Rq com distribuição gaussiana e d
ou não.
Ao polinómio recebido como argumento, aplicamos uma transformação p
decrescente.
Depois fazemos a soma dos **h** maiores elementos e verificamos se
parâmetros **L_E**.
'''
def checkE(e):
    my_sum = 0
    e_list = list(e)
    e_list.sort(reverse=True)
    for i in range(0,h):
        my_sum += e_list[i]

    if my_sum > L_E:
        return 1
    return 0


'''
generate_S(): Devolve um polinómio aleatório em Rq, com distribuiçã
Este é gerado através da função pré-definida $$ random_element() $$
Ao polinómio gerado é aplicado um teste de limites através da funçã
'''
def generate_S():
    s = None
    while True:
        s = Rq.random_element(distribution="gaussian")
        if checkS(s) == 0:
            return s


'''
checkS(s): Recebe um polinómio em Rq com distribuição gaussiana e d
ou não.
Ao polinómio recebido como argumento, aplicamos uma transformação p
decrescente.
Depois fazemos a soma dos **h** maiores elementos e verificamos se
parâmetros **L_S**.
'''
```

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

    if my_sum > L_S:
        return 1
    return 0
```

Função key_generation

Esta é a função resonsável pela geração das chaves públicas e secretas.

Esta versão é uma versão simplificada da apresentada no documento do esquema submetido ao concurso NIST, visto que estamos a implementar no SageMath. Esta simplificação passou por não recorrer às seeds(tal como era sugerido no documento) e recorrermos aos polinómios e a alguns métodos que os anéis oferecem.

Assim começamos por gerar o polinómio `a` que é aleatóriamente gerado através do anel (como foi mostrado na implementação acima) , assim como os polinómios `s` e `e`.

Depois calculamos então o polinómio `t` representado: `t = a*s + e`.

Por fim construímos as chaves secretas e privadas:

- **sk** - (a,e,s)
- **pk** - (t,a)

In [5]:
```
'''
key_generation(): função que gera as chaves públicas e secretas.
A chave secreta é composta pelos polinómios: ( a , e , s )
A chave privada é composta pelos polinómios: ( t , a )
'''

def key_generation():
    print('\n--- Starting KeyGen Process ---\n')

    a = Gen_A()
    #print(a)
    print('We have **polynomial a**')

    s = generate_S()
    print('We have **s**')

    e = generate_E()
    print('We have **e**')

    t = a*s + e
    print('We have **t**')

    sk = (a,e,s)
    #print(sk)
    print('We have **sk**')
    pk = (t,a)
    #print(pk)
    print('We have **pk**')

    print('\n--- KeyGen Process Done ---\n')

    return sk,pk

(sk,pk) = key_generation()
```

```
--- Starting KeyGen Process ---

We have **polynomial a**
We have **s**
We have **e**
We have **t**
We have **sk**
We have **pk**

--- KeyGen Process Done ---
```

Métodos auxiliares

Para os processos de assinatura e verificação vamos precisar de alguns métodos e de transformação. Assim nesta célula vamos encontrar a implementação desses métodos.

In [6]:

```
'''
lift_at_center(x) recebe como argumento um x e faz lift dele.
'''
def lift_at_center(x):
    center = q//2
    return lift(x+center) - center

'''
poly_round_to_center(poly) recebe como argumento um polinómio e ret
transformados com a função $$ lift_at_center $$
'''
def poly_round_to_center(poly):
    coefs = poly.list()
    ret = []
    for coef in coefs:
        ret.append(Zx(lift_at_center(coef)))
    return ret

from sage.cpython.string import str_to_bytes
'''
hash(m) recebe uma mensagem m (em string) e calcula a hash da mesma
'''
def hash(m):
    new_m = str_to_bytes(m)
    h = hashlib.sha256()
    h.update(new_m)
    return h.digest()

from sage.crypto.util import ascii_integer
'''
something_to_bin(inputs) recebe como argumento um input qualquer e
'''
def something_to_bin(inputs):
    bin = BinaryStrings()
    return list(bin.encoding(str(inputs)))

'''
H(v,hash_m) recebe como argumento um polinómio v e a hash de uma me
Com o polinómio v calculamos a lsita binária v; com o hash_m calcul
o par (w,hash_m_list)
'''
def H(v,hash_m):
    w = [None] * n
    for i in range(n):
        val = v[i] % 2^d
        #val = mod(v[i],2^d)
        if(val > 2^(d-1)):
            val = val - 2^d
        w[i] = (v[i] - val)/ (2^d)
    hash_m_list = something_to_bin(hash_m)
    return (w,hash_m_list)
```

Assinar uma mensagem

Funções auxiliares

De forma a procedermos à assinatura de uma mensagem decidimos criar uma função auxiliar que calcula um dos parâmetros necessários para a assinatura.

In [7]:
```
'''
calculate_Z(y,s,c1) recebe como argumento 2 polinómios (em Rq) e um
Com a lista binária aplicamos uma transformação de modo a obtermos
** z = y + s*c**
'''

def calculate_Z(y,s,c1):
    c = Rq([c1[i] for i in range(n)])
    return (y + s*c)
```

Sign_message

Para assinarmos uma mensagem vamos precisar da mensagem e da chave secreta.

Depois começamos por criar um polinómio com distribuição uniforme e localizado no intervalo [-B,B].

Depois multiplicamos os polinómios `a` e `y` e aplicamos um round_to_center a esse novo polinómio (obtendo assim `v`).

Por fim calculamos `c=(c1,c2)` que pode ser obtido da função auxilar `H(v,hash_m)` que receberá como argumento o polinómio calculado e a hash da mensagem.

A partir da primeira componente de `c` e dos polinómios `y` e `s` calculamos o novo polinómio `z`.

A assinatura é então composta pelos polinómios `z` e `c` e ainda pela hash da mensagem m.

In [8]:
```
'''
sign_message(m,sk): recebe como argumento a mensagem e a chave secr
'''

def sign_message(m,sk):
    print('\n--- Starting Signing Message Process ---\n')

    (a,e,s) = sk

    y = Rq.random_element(x=-B, y = B+1, distribution='uniform')
    print('We have **y**')

    v = poly_round_to_center(a*y)
    print('We have **v**')


    (c1,c2) = H(v,hash(m))
    c = (c1,c2)
    print('We have **c**')

    z = calculate_Z(y,s,c1)
    print('We have **z**')

    signature = (z,c)
    print('We have **signature**')

    print('\n--- Signing Message Process Done ---\n')
    return signature

signature = sign_message('test',sk)
#print(signature)
```

```
--- Starting Signing Message Process ---

We have **y**
We have **v**
We have **c**
We have **z**
We have **signature**

--- Signing Message Process Done ---
```

Verificar assinatura

Funções auxiliares

Tal como para a assinatura foi necessário uma função auxiliar que nos calculasse o parâmetro `w` .

In [9]:
```
'''
Recebe como parâmetros os polinómios *a* e *t* e a lista de coefici
A partir das listas calculamos os respetivos polinómios. Depois faz
operação: a*z + t*c
'''

def calculate_W(a,t,z,c):
    poly_z = Rq([z[i] for i in range(n)])
    poly_c = Rq([c[i] for i in range(n)])

    left = a*poly_z
    right = t*poly_c

    w = poly_round_to_center(left-right)
    return w
```

Verify_signature

De forma a verificarmos uma assinatura vamos precisar da assinatura, da chave pública e da mensagem.

Começamos por partir a assinatura e a chave pública nas respetivas componentes, assim como a componente `c` no par `(c1,c2)`.

A partir daí calculamos o polinómio `w` com a função auxiliar `calculate_W(a,t,z,c1)` recebendo os polinómios `a`, `t` e as listas de coeficientes `z` e `c1`.

Com esse novo polinómio `w` aplicamos a função `H()` com o polinómio de `w` e hash da mensagem `m` como argumentos.

Com esse resultado podemos comparar c1 com a primeira componente do par resultante da operação em cima. Se estas forem iguais, a assinatura é válida (e inválida caso contrário).

In [10]:
```
'''
verify_signature(m,signature,pk) recebe como argumento a mensagem m
vericidade da assiantura para a dada mensagem m.
'''
def verify_signature(m, signature, pk):
    print('\n--- Starting Signature Verification Process ---\n')

    (z,c) = signature
    (t,a) = pk
    (c1,c2) = c

    w = calculate_W(a,t,z,c1)

    print('We have **w**')

    (hw1,hw2) = H(w,hash(m))


    if c1!=hw1:
        print('\nError in signature.')
        print('\n--- Signature Verification Process Done ---\n')
        return False

    else:
        print('\nSignature is valid.')
        print('\n--- Signature Verification Process Done---\n')
        return True




verify_signature('test', signature, pk)
```

```
--- Starting Signature Verification Process ---

We have **w**

Error in signature.

--- Signature Verification Process Done ---
```

Out[10]:
```
False
```

In [ ]:
```

```