TP3 - qTesla

June 7, 2021

1 qTesla - grupo 6 - Ana Margarida Campos (A85166) , Nuno Pereira (PG42846)

Neste exercício foi proposta a implementação do esquema ***qTesla***, um esquema de assinatura digital pós-quântica candidato ao concurso *NIST-PQC*, neste caso a segunda ronda deste concurso. A segurança deste, baseia-se no R-LWE (*Ring Learning with errors*). De seguida é apresentada e descrita o esquema onde são revelados os passos seguidos para a resolução do exercício bem como uma explicação detalhada de cada função elaborada. Esta versão foi desenvolvida tendo por base o documento *qTESLA_roud2_12.02.2020.pdf*.

1.1 Proposta de resolução

Os três principais passos neste esquema de assinatura digital pós-quântica são a geração das chaves pública e privada, a assinatura da mensagem e, por fim, a verificação da assinatura. Para que estes passos sejam realizados com sucesso, é necessário recorrer a algumas funções auxiliares. Estas são:

- ***PRF1***: *pseudorandom function*, utilizando SHAKE-128 que recebe uma *pre-seed* com *k bits* e mapeia em *K+3 seeds* cada uma com *k bits*. Todas as *seeds* necessárias durante a geração das chaves são obtidas a partir desta função;
- ***PRF2***: *pseudorandom function*, utilizando SHAKE-128 que recebe como *input* uma *seedy*, um valor aleatório *r* e a *hash G* da mensagem e mapeia os mesmos numa *seed* denominada *rand* com *k bits*;
- ***checkS***: recebe como *input* um polinómio secreto e verifica se a soma dos maiores valores dos coeficientes é suporios a *LS*. É usada para simplificar a redução da segurança;
- ***checkE***: recebe como *input* um polinómio de erro e verifica se a soma dos maiores valores dos coeficientes é suporios a *LE*. Garante a correção do esquema de assinatura;
- ***GenA***: é uma função geradora de polinómios públicos que recebe uma *seed* com *k-bit* e mapeia em *K* polinómios *Rq*;
- ***G***: função de *hash* resistente a colisões que utiliza SHAKE-128;
- ***ySampler***: recorre aos argumentos *seed rand* e *nounce counter* de forma a criar um polinómio pertencete a *R*;
- ***Enc***: realiza a codificação de um *hash* para polinómio. Este polinómio é representado por dois arrays *pos_list* e *sign_list*. O primeiro contém as posições e o segundo contém os sinais dos coeficientes não zero;

- ***H***: recebe como *input* K polinómios pertencentes a Rq e como resultado dá um *hash* com k *bits* dos polinómios juntamente com o *hash* G da mensagem e o *hash* G de t.
- ***sparse_multiplication***: maneira eficiente de realizar a multiplicação de polinómios;
- ***test_rejection***: verificação de que os coeficientes de um polinómio se encontram entre o intervalo [-B, B] durante a assinatura da mensagem.

Para além destas funções, recorremos à função ***NTT*** disponibilizada pelo docente e à função ***cSHAKE128***, a qual, foi retirada de um repositório do *gitHub*. Esta segunda função encontra-se no ficheiro *cshake.py*. Após a implementação das funções explicadas anteriormente, foram desenvolvidas as funções principais. Estas são descritas de seguida:

- ***key_generation***: consiste na geração das chaves pública e privada. Para tal, inicialmente, são gerados os polinómios públicos com a utilização da função ***GenA*** através da expanção de uma *seed* usando *PRF1*. Posteriormente, é gerado um polinómio secreto a partir da distribuição gaussiana e ocorre a verificação utilizando as funções ***checkS*** e ***checkE***. Se forem cumpridos estes requisitos, as chaves pública e privada são geradas.
- ***signature_gen***: esta função consiste na assinatura da mensagem recebida como argumento, recorrendo à chave privada criada anteriormente. Desta forma, é primeiramente gerado uniformemente de forma aleatória um polinómio y através da função ***ySampler***. Esta recebe como *input* uma *string* aleatória criada a partir da função ***PRF2*** para fornecer aleatoriedade. Recorremos à função ***GenA*** de maneira a criar os polinómios públicos que serão necessários para o cálculo dos polinómios *v[i]*y mod +- q*. Um dos parâmetros da assinatura, c, é gerado a partir da função ***H*** que recebe os polinómios v criados anteriormente, a *hash* da mensagem e g pertencente à chave secreta. Para criar o outro parâmetro da assinatura, z, recorre-se às funções ***Enc*** e ***sparce_multiplication***. É necessário que os testes de segurança e correção sejam realizados, no entanto, nesta implementação, esses testes não foram possíveis de realizar devido a erros de implementação.
- ***verification***: função utilizada para verificar a assinatura da mensagem. Com este propósito, primeiramente são gerados os polinómios públicos com recurso à função ***GenA***. É calculado um vetor w e depois, com vista a verificar a assinatura, calcula-se o *hash* c de w, *G(m)* e *G(t)*. Ocorre a verificação deste com o valor recebido como parâmetro. Se for igual, então a assinatura é válida. Caso contrário, a assinatura é inválida e é rejeitado.

Como é possível verificar através do *output* recebido após a execução do programa, a assinatura encontra-se inválida. Tal se pode dever ao facto dos testes de segurança e correção não terem sido corretamente implementados.

[1]:
```
#imports necessários para a implementação
from cryptography.hazmat.primitives import hashes
from cshake import cSHAKE128
from sage.stats.distributions.discrete_gaussian_polynomial import␣
 ↪DiscreteGaussianDistributionPolynomialSampler
import pickle
import math
import numpy
```

[2]:
```
class NTT(object):
#
    def __init__(self, n=128, q=None):
        if not  n in [32,64,128,256,512,1024,2048]:
            raise ValueError("improper argument ",n)
        self.n = n
        if not q:
            self.q = 1 + 2*n
            while True:
                if (self.q).is_prime():
                    break
                self.q += 2*n
        else:
            if q % (2*n) != 1:
                raise ValueError("Valor de 'q' não verifica a condição NTT")
            self.q = q

        self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen()

        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)])

    def random_pol(self,args=None):
        return (self.R).random_element(args)
```

[27]:
```
# Constantes qTesla - categoria de segurança 1 do NIST
k = 256 #k
n = 1024
K = 4 #kk
q = 343576577
h = 25
LE = 554
LS = 554
S = LS
E = LE
B = 2**19 - 1
d = 22
bgenA = 108
rate_xof = 168
chunk_size = 512
beta = 64
t = 78
ntt = NTT(n,q)

_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))

# FUNÇÕES AUXILIARES

# pseudorandom function, utilizando SHAKE-128
def PRF1(pre_seed):
    digest = hashes.Hash(hashes.SHAKE128(int(k * (K + 3)/8)))
    digest.update(pre_seed)
    seed = digest.finalize()
    seed_s = seed[0:32]
    seed_e1 = seed[32:64]
    seed_e2 = seed[64:96]
    seed_e3 = seed[96:128]
    seed_e4 = seed[128:160]
    seed_e = [seed_e1, seed_e2, seed_e3, seed_e4]
    seed_a = seed[160:192]
```

```
    seed_y = seed[192:224]
    return seed_s, seed_e1, seed_e2, seed_e3, seed_e4, seed_a, seed_y, seed_e

# pseudorandom function, utilizando SHAKE-128
def PRF2(seed_y, r, G):
    res = seed_y + r + G
    digest = hashes.Hash(hashes.SHAKE128(int(k/8)))
    digest.update(res)
    seed = digest.finalize()
    return seed

# verificação de que ||s||<LS
def checkS(s):
    ls = s.list()
    ls.sort(reverse=True)
    soma = 0
    for i in range(h):
        soma += ls[i]
    if soma>LS:
        return 1
    return 0

# verificação de que ||e||<LE
def checkE(e):
    le = e.list()
    soma = 0
    le.sort(reverse=True)
    for i in range(h):
        soma += le[i]
    if soma>LE:
        return 1
    return 0

# função geradora de polinómios públicos
def GenA(seed_a):
    D = 0
    b = ceil(log(q,2)/8) #ceil arredonda para cima
    b1 = bgenA
    cT = cSHAKE128(seed_a, rate_xof*b1, bytes([D]), '')
    c = [0]*int((rate_xof * b1)/(b*8))
    for i in range(int((rate_xof * b1)/(b*8))):
        c[i] = cT[i*b:(i+1)*b]
    i = 0
    pos = 0
    a = [[ 0 for x in range(n)] for y in range(K+1)]
    while i<(K*n):
        if pos>int((floor((rate_xof * b1)/(b*8))-1)):
```

```
            D = D + 1
            pos = 0
            b1 = 1
            c1 = cSHAKE128(seed_a, rate_xof * b1, int(D).
 ↪to_bytes(2,byteorder="big"), '')
            for j in range(int((rate_xof * b1)/(b*8))):
                c[j] = c1[j*b:(j+1)*b]
        if mod(int.from_bytes(c[pos],"big"),2 ** (ceil(log(q,2)))) < q:
            a[floor(i/n)+1][i-n*floor(i/n)] = mod(int.
 ↪from_bytes(c[pos],"big"),2 ** (ceil(log(q,2))))
            i = i + 1
        pos = pos + 1
    a.pop(0)
    aAux=[]
    for i in range(K):
        aAux.append(Rq(a[i]))
    return aAux

# função de hash resistente a colisões que utiliza SHAKE-128
def G(t):
    digest = hashes.Hash(hashes.SHAKE128(int(40)))
    digest.update(t)
    seed = digest.finalize()
    return seed

# recorre aos argumentos rand e D de forma a criar um polinómio pertencete a R
def ySampler(rand, D):
    pos = 0
    n1 = n
    D1 = D * 2**8
    c = [0]*n
    y = [0]*n
    b = ceil(log((B+1),2)/8)
    cT = cSHAKE128(rand, b*n1, int(D1).to_bytes(2,byteorder="big"), '')
    for w in range(n):
        c[w] = cT[w*b:(w+1)*b]
    i = 0
    while i<n:
        if pos>=n1:
            D = D + 1
            pos = 0
            n1 = floor(rate_xof/b)
            cT2 = cSHAKE128(rand, b*n1, int(D1).to_bytes(2,byteorder="big"), '')
            for w in range(n):
                c[w] = cT2[w*b:(w+1)*b]
        y[i] = mod(int.from_bytes(c[pos],"big"), 2**(ceil(log(B,2)) +1) -B)
        if y[i] != B +1:
```

```
            i = i + 1
        pos = pos + 1
    return R(y)

# realiza a codificação de um hash para polinómio.
# Este polinómio é representado por dois arrays pos_list e sign_list.
# O primeiro contém as posições e o segundo contém os sinais dos coeficientes␣
 ↪não zero.
def Enc(c1):
    D = 0
    cnt = 0
    c = [0]*n
    r = [0]*rate_xof
    pos_list = [0] * h
    sign_list = [0] * h
    rT = cSHAKE128(c1, rate_xof, int(D).to_bytes(2,byteorder="big"), '')
    for u in range(rate_xof):
        r[u] = rT[u*1: (u*1) + 1]
    i = 0
    while i < h:
        if cnt > (rate_xof-3):
            D = D + 1
            cnt = 0
            rT = cSHAKE128(c1, rate_xof, int(D).to_bytes(2,byteorder="big"), '')
            for u in range(rate_xof):
                r[u] = rT[u*1: (u*1) + 1]
        pos = mod(int.from_bytes(r[cnt],"big") * 2**8 + int.from_bytes(r[cnt +␣
 ↪1],"big"), n)
        if c[pos] ==0 :
            if mod(int.from_bytes(r[cnt +2],"big"),2) == 1:
                c[pos] = -1
            else:
                c[pos] = 1
            pos_list[i] = pos
            sign_list[i] = c[pos]
            i = i + 1
        cnt = cnt + 3
    return pos_list, sign_list


# multiplicação de polinómios utlizando ntt
def ntt_multiplication(a, s):
    a_ntt = ntt.ntt(a)
    s_ntt = ntt.ntt(s)
    a_s = [x * y for x, y in zip(a_ntt, s_ntt)]
    return ntt.ntt_inv(a_s)
```

```
# dá um hash com k bits
def H(v, g_m, g_t):
    w = [0] * (k*n+80)
    for i in range(K):
        for j in range(n):
            val = int(mod(v[i][j], 2**d))
            if val>2**(d-1):
                val = val - (2**d)
            w[(i-1)*n+j] = (int(v[i][j]) - val)/(2**d)
    w[K*n:K*n+40] = g_m
    w[K*n+40:K*n+80] = g_t
    digest = hashes.Hash(hashes.SHAKE128(int(k/8)))
    digest.update(pickle.dumps(w))
    c = digest.finalize()
    return c

# cálculo do mod +-
def _mod(p,q):
    lista = []
    for i in p:
        j = int(i) % q
        j -= math.floor(q/2)
        lista.append(j)

    return lista

# maneira eficiente de realizar a multiplicação de polinómios
def sparse_multiplication(g, pos_list, sign_list):
    f = [0] * n
    for i in range(h):
        pos = pos_list[i]
        for j in range(pos):
            f[j] = f[j] - sign_list[i] * g[j+n-pos]
        for j in range(pos, n):
            f[j] = f[j] + sign_list[i] * g[j-pos]
    return Rq(f)


# verificação de que os coeficientes de um polinómio se encontram entre o␣
 ↪intervalo [-B, B]
def test_rejection(z):
    valid = numpy.uint32(0);
    for i in range(n):
        valid |= numpy.uint32(B-S) - abs(numpy.int32(z[i]));
    return (int)(valid >> 31);

# geração das chaves pública e privada
```

```
def key_generation():
    e = [0]*k
    counter = 1
    pre_seed = os.urandom(32)
    seed_s, seed_e1, seed_e2, seed_e3, seed_e4, seed_a, seed_y, seed_e =␣
 ↪PRF1(pre_seed)
    a = GenA(seed_a)
    s = DiscreteGaussianDistributionPolynomialSampler(R, n, 3.0)()
    while checkS(s) != 0:
        s = DiscreteGaussianDistributionPolynomialSampler(R, n, 3.0)()
        counter = counter + 1
    for i in range(K):
        e[i] =DiscreteGaussianDistributionPolynomialSampler(R, n, 3.0)()
        while checkE(e[i]) != 0:
            e[i] = DiscreteGaussianDistributionPolynomialSampler(R, n, 3.0)()
            counter = counter + 1
        t = ntt_multiplication(a[i].lift(), s.lift()) + e[i]
    g = G(pickle.dumps(t))
    sk = (s,e[0], e[1], e[2], e[3],seed_e, seed_a, seed_y, g)
    pk = (t, seed_a)
    return sk, pk

# assinatura da mensagem passada como argumento
def signature_gen(m, sk):
    s, e_0, e_1, e_2, e_3, seed_e, seed_a, seed_y, g = sk
    counter = 1
    r = os.urandom(k)
    w = [[0 for i in range(n)] for i in range(k)]
    rand = PRF2(seed_y, r, G(m))
    y = ySampler(rand, counter)
    a = GenA(seed_a)
    v = []
    w = [[0 for i in range(n)] for i in range(k)]
    for i in range(K):
        mult = ntt_multiplication(a[i].lift(), y.lift()).list()
        v.append(_mod(mult,q))
    c = H(v,G(m),g)
    pos_list, sign_list = Enc(c)
    sparse = sparse_multiplication(s, pos_list, sign_list)
    z = y + sparse
    return z, c
    # testes de segurança e correção que não forem possíveis de implementar
    '''while test_rejection(z)== -1:
        counter = counter + 1
        y = ySampler(rand, counter)
        a = GenA(seed_a)
        v = []
```

```
        for i in range(K):
            mult = ntt_multiplication(a[i].lift(), y.lift()).list()
            v.append(_mod(mult))
        c = H(v,G(m),g)
        pos_list, sign_list = Enc(c)
        sparse = sparse_multiplication(s, pos_list, sign_list)
        z = y + sparse

    for i in range(K):
        sp = int.from_bytes(c, "big")  * int.from_bytes(seed_e[i], "big")
        print("vi===", v[i])
        print("sp==", sp)
        w[i] = _mod(v[i] - sp)
    print("w===", w)
    '''

# verificação da assinatura
def verification(m, z, c, pk):
    t, seed_a = pk
    pos_list, sign_list = Enc(c)
    a = GenA(seed_a)
    w = [0]*4
    for i in range(K):
        smult = sparse_multiplication(Rq(t[i]), pos_list, sign_list)
        w[i] =  ntt_multiplication(a[i], z) - Rq(_mod(smult,q))
    if c != H(w, G(m), G(pickle.dumps(t))):
        print("Assinatura inválida")
        return -1
    else:
        print("Assinatura válida")
        return 0
```

[28]:
```
sk, pk = key_generation()
m = os.urandom(32)
z, c = signature_gen(m, sk)
v = verification(m,z, c, pk)
```

```
Assinatura inválida
```