qTESLA

June 7, 2021

1 TP3 - Estruturas Criptográficas

1.1 Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

2 qTESLA

2.0.1 Criação dos parâmetros

[75]:
```
import os
import math
import sys
import hashlib
from cryptography.hazmat.primitives import hashes, hmac
from sage.crypto.util import ascii_integer
from sage.stats.distributions.discrete_gaussian_polynomial import␣
↪DiscreteGaussianDistributionPolynomialSampler


class QTesla:
    def __init__(self):                          #Criação dos parâmetros
        self.lambeda = 95
        self.capa = 256
        self.n = 1024
        self.sigma = 8.5
        self.k = 4
        self.q = 343576577
        self.h = 25
        self.Le = 554
        self.Ls = 554
        self.B = (2^19)-1
        self.d = 22


        Q = GF(self.q)
        Fq.<y> = Q[]
```

```
        self.Rq = Fq.quotient(y^self.n + 1)

        Zx.<x> = ZZ[]
        self.R = Zx.quotient(x^self.n + 1)
```

2.0.2 Criação das chaves

[68]:
```
class QTesla(QTesla):

    # Função auxiliar para criação da matriz A
    def genA(self):
        Fq = GF(self.q)
        Fqy.<y> = Fq[]
        Rq.<y> = Fqy.quotient(y^self.n + 1)
        return Rq.random_element()

    # Verifica se a soma dos h maiores elementos é menores que o limite␣
↪definido nos
    # parâmetros Le.
    def checkE(self, e):
        E = e.list()
        for i in range(len(E)):
            E[i] = abs(int(E[i]))
        E.sort()
        E = E[self.n-self.h:]
        if sum(E) >= self.Le:
            return 1
        else:
            return 0

    # Verifica se a soma dos h maiores elementos é menores que o limite␣
↪definido nos
    # parâmetros Ls.
    def checkS(self,s):
        S = s.list()
        for i in range(len(S)):
            S[i] = abs(int(S[i]))
        S.sort()
        S = S[self.n-self.h:]
        if sum(S) >= self.Ls:
            return 1
        else:
            return 0
```

```
    # Função que gera as chaves
    def keygen(self):
        counter = 1
        A = []
        e = [None] * self.k
        s = None
        t = []

        # Criação da matriz A de polinómios em Rq
        for i in range(self.k):
            A.append(self.genA())

        # Criação polinómio s, de acordo com uma distribuição gaussiana
        while(True):
            s = DiscreteGaussianDistributionPolynomialSampler(self.R, self.n,␣
↪self.sigma)()
            counter += 1
            if(self.checkS(s) == 0):
                break
        # Criação vetor de polinómios e, de acordo com uma distribuição␣
↪gaussiana
        for i in range(self.k):
            while(True):
                e[i] = DiscreteGaussianDistributionPolynomialSampler(self.R,␣
↪self.n, self.sigma)()
                counter += 1
                if(self.checkE(e[i]) == 0):
                    break
            t.append(A[i] * self.Rq(s.lift())+ self.Rq(e[i].lift()))

        sk = (s, e, A)
        pk = (t,A)
        k = (sk,pk)

        print("Chaves geradas!")

        return sk, pk
```

2.1 Sign

[72]:
```
class QTesla(QTesla):

    # Função de Hash H que vai dar "commit" pelos "high bits"
    def H(self,v,hash_m):
        Zx.<x>  = ZZ[]
```

```
        w = bytearray()
        for i in range(0,(self.k)):
            vv = v[i].list()
            poly = bytearray()
            for j in range(self.n):
                val = vv[j] % 2^self.d
                if(val > 2^(self.d-1)):
                    val = val - 2^self.d

                val_byt = bytes(abs(val))
                poly += val_byt
            w+= poly

        cl = hashlib.shake_128(w).digest(int(self.capa/8))
        return cl

    # Codifica o desafio num polinómio
    def enc(self, cl):
        rate_xof = 168
        D = 0
        cnt = 0
        t= 168
        r = [168] * t

        for i in range(t):
            x=hashlib.shake_128(cl).digest(int(8))
            a  = int.from_bytes(x,byteorder='big')
            r[i] = a

        i = 0
        c = [0] * self.n
        pos_list = [0] * self.h
        sign_list = [0] * self.h
        while i < self.h:
            if cnt > (rate_xof - 3):
                D = D + 1
                cnt = 0
                for i in range(t):
                    x=hashlib.shake_128(cl).digest(int(8))
                    a  = int.from_bytes(x,byteorder='big')
            r[i] = a

            pos = (r[cnt] * 2**8 + r[cnt+1]) % self.n
            if c[pos] == 0:
                if (r[cnt+2] % 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)


# Calcula norma ifinita
def infinity_norm(self, val):
    tam = len(val)
    temp = 0
    for i in range(tam):
        if abs(f[i]) > temp:
            temp = abs(f[i])
    return temp

# Multiplicação
def sparse_mult(self, g, pos_list, sign_list):
    f = [0] * self.n
    gg = g.list()

    for i in range(self.h):
        pos = pos_list[i]
        for j in range(pos-1):

            f[j] = f[j] - sign_list[i] * gg[j+self.n-pos]
        for j in range(self.n -1 ):
            f[j] = f[j] + sign_list[i] * gg[j-pos]
    return f


 # Norma infinita
def infinite_norm(self,p):
    #R = self.pol_aux(p,n)
    J = p.list()
    for i in range(len(J)):
        J[i] = abs(int(J[i]))
    return int(max(J))


# Função para assinar uma mensagem
def sign(self, m ,sk):
    s = sk[0]
    e = sk[1]
    A = sk[2]
```

```
        counter = 1
        r = os.urandom(self.capa)

        # Gera polinómio y
        y = self.Rq.random_element(x=-self.B, y = self.B+1,␣
↪distribution='uniform')


        while True:
            v = [0] * self.n
            z = [0] * self.n
            w = [0] * self.n

            for i in range(self.k):
                a = A[i] * y
                v[i] = a.lift().map_coefficients(lambda c: c.lift_centered(),␣
↪ZZ)

            # Commit e obtenção do desafio
            cl = self.H(v, hashlib.shake_256(m).digest(int(40)))
            (pos_list, sign_list) = self.enc(cl)
            c = (pos_list, sign_list)

            # Multiplicação s*c
            sc = self.sparse_mult(s,pos_list,sign_list)

            # Calculo de z
            z = y + self.Rq(sc)

            flag = True

            # Asegurar segurança ("rejection sampling")
            """
            if self.well_rounded(self.Le,z) == False:
                flag = False
                print("here")
                break
            """

            if flag == False:
                continue

            flagl = True
            # Assegurar correção
            for i in range(1, self.k+1):

                ec = self.Rq(self.sparse_mult(e[i],pos_list,sign_list))
```

```
                w[i] = v[i] - ec

                if (self.infinite_norm(w[i]) >= 2**(self.d-1)-self.Le) or (self.
↪infinite_norm(w[i]) >= (int(math.floor(q/2)) - self.Le)):
                    counter = counter + 1
                    flag = False
                    break
            if flagl == True:
                return (z, cl)
```

2.2 Verify

[73]:
```
class QTesla(QTesla):

    # Função para verificação da assinatura
    def verify(self, m, signature, pk):
        (z, cl) = signature
        t, A = pk

        (pos_list, sign_list) = self.enc(cl)
        w = [0] * self.n
        for i in range(self.k):
            tc = self.Rq(self.sparse_mult(t[i],pos_list,sign_list))

            res = A[i]*z - tc
            w[i] = res.lift().map_coefficients(lambda c: c.lift_centered(), ZZ)


        c_ = self.H(w, hashlib.shake_256(m).digest(int(40)))

        if cl != c_:
            print('Assinatura Inválida')
        else:
            print('Assinatura Válida')
```

[74]:
```
q = QTesla()

m = b'Ola'
sk,pk = q.keygen()
sig = q.sign(m,sk)
q.verify(m,sig,pk)
```

```
Chaves geradas!
Assinatura Inválida
```

[ ]:

[ ]: