

Grupo12-Dilithium-TP3

May 30, 2022

1 Trabalho prático 3 - Estruturas Criptográficas

Autores: Arianas Lousada (PG47034), Cláudio Moreira (PG47844)

Grupo 12

1.1 Dilithium

Para o presente trabalho prático foi proposta a implementação do esquema de assinatura digital *Dilithium*, cuja segurança depende da dificuldade de encontrar vetores curtos em reticulados.

As três principais funcionalidades do presente esquema de assinatura digital correspondem a: criação das chaves pública e privada, da assinatura da mensagem e a verificação da mesma. Para isto, foram desenvolvidas algumas funções auxiliares:

- ***H***: função de *hash* que recorre ao *SHAKE256* para construir um array com 256 elementos de -1 a 0, com a particularidade de existirem τ coeficientes iguais a -1 e 1 e os restantes iguais a 0;
- ***sample***: gera um vetor aleatório onde cada coeficiente desse vetor é um elemento de R_q ;
- ***decompose***: extrai bits de *higher-order* e *lower-order* de elementos de Z_q ;
- ***highBits***: recupera os bits de ordem superior. Para tal extrai r_1 do *output* da função ***decompose***;
- ***lowBits***: recupera os bits de ordem inferior, com raciocínio semelhante à função anterior. Para tal extrai r_0 do *output* da função ***decompose***;
- ***infinity_norm***: função utilizada para calcular a norma infinita de um array de um determinado polinómio.

Com isto, foram construídas 3 funções principais de modo a atingir as 3 funcionalidades anteriormente mencionadas:

- ***gen***: consiste na geração das chaves pública e privada. Para atingir este fim, começa-se por gerar uma matriz A , $k \times l$, com polinómios aleatórios pertencentes ao anel R_q . Posteriormente, recorrendo à função ***sample***, são gerados dois vetores de chave secretos aleatórios. Estes vão ser parte da chave privada. Por fim, de modo a calcular a segunda parte da chave pública, é efetuado o cálculo $t = As_1 + s_2$. Ambas as chaves contêm a matriz A e t .
- ***sign***: esta função consiste na assinatura da mensagem recebida como argumento, recorrendo à chave privada criada anteriormente. Para tal, gera-se inicialmente um vetor y cujo número

de coeficientes tem de ser inferior a $y1$ (número suficientemente alto para impedir a revelação da chave secreta e suficientemente pequeno para que a assinatura não seja falsificada). Posteriormente é realizada a multiplicação da matriz A com o vetor y e é criado um vetor $w1$ com todos os *high bits* de Ay . É também criada a *hash* (c) da mensagem com $w1$ recorrendo à função auxiliar **H**. Uma potencial assinatura é gerada através de $z = y + cs1$. No entanto, esta tem de respeitar duas restrições de modo a que o esquema não seja inseguro. Estas restrições implicam que a norma infinita de z seja inferior a $y1 - \beta$ e que a norma infinita de $LowBits(Ay - cs2, 2y2)$ seja inferior a $y2 - \beta$. Por fim, é retornado z e c que correspondem à assinatura.

- **verify**: função utilizada para verificar a assinatura da mensagem. Recebe como argumentos a mensagem, chave pública e assinatura. É apenas gerado o $w1$ (da mesma forma que foi gerado na função **sign**) e, através deste, calculado o *hash*. Se o *hash* calculado for igual ao da assinatura e se a norma infinita de z for inferior a $y1 - \beta$, então a assinatura é válida. Caso contrário é inválida.

```
[1]: # imports necessários para a implementação
from cryptography.hazmat.primitives import hashes
import numpy as np
import pickle
import random
from numpy import linalg as LA
```

```
[2]: # constantes Dilithium NIST Security Level = 2
q = 8380417
n = 256
d = 13
T = 39
beta = 78
k = 4
l = 4
secret_key_range = 2
y1 = 2**17
y2 = (q-1)/88
h = 60

# definição de aneis
Zx.<w> = ZZ[]
R.<x> = QuotientRing(Zx,Zx.ideal(w^n+1))

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

# Funções auxiliares

# função de conversao: bytes -> bits
# utilizada na função H
def bytes_to_bits(s):
```

```

s = s.decode('ISO-8859-1')
result = []
for c in s:
    bits = bin(ord(c))[2:]
    bits = '00000000'[len(bits):] + bits
    result.extend([int(b) for b in bits])
u=""
for i in result:
    u = u + str(i)
return u

# criação de um array com 256 elementos que contenha t
# coeficientes iguais a -1 e 1 e o resto zeros
# utiliza o SHAKE256
def H(a):
    digest = hashes.Hash(hashes.SHAKE256(int(32)))
    digest.update(a)
    c = digest.finalize()
    bits = bytes_to_bits(c)
    counter = 0
    res = []
    for i, v in enumerate(bits):
        if i > T:
            res.append(0)
            continue
        if v == '0':
            res.append(-1)
            continue
        else:
            res.append(1)
            continue

    return res

# gera um vetor aleatorio no qual cada coeficiente
# é um elemento do anel  $R_q$ 
def sample(key_range, max_size):
    lista = []
    for i in range(max_size):
        p = []
        for j in range(n):
            p.append(randint(1, key_range))
        lista.append( $R_q(p)$ )
    return lista

# extrai bits de higher-order e lower-order de elementos pertencentes a  $Z_q$ 

```

```

def decompose(r, alpha):
    r = int(mod(r,q))
    r0 = int(mod(r, alpha))
    if r-r0 == q-1:
        r1 = 0
        r0 = r0-1
    else:
        r1 = (r-r0)/alpha
    return r1, r0

# extrai r1 do output da funcao anterior (decompose)
def highBits(r, alpha):
    r1, r0 = decompose(r, alpha)
    return r1

# extrai r0 do output da funcao decompose
def lowBits(r, alpha):
    r1, r0 = decompose(r, alpha)
    return r0

# norma infinita: coeficiente absoluto máximo do array de polinômios
def infinity_norm(z):
    norma = 0
    lista = []
    for i in z:
        for j in i:
            for v in j:
                lista.append(abs(int(v)))
    maior = max(lista)
    if(maior>norma):
        norma = maior
    return norma

# funcao que gera o par (publicKey, privatekey)
def gen():
    Aaux = []
    for i in range(k*1):
        Aaux.append(Rq.random_element())
    A = matrix(Rq, k, 1, Aaux)
    s1 = sample(secret_key_range, 1)
    s2 = sample(secret_key_range, k)
    tTemp = matrix(A) * matrix(Rq, 1, 1, s1)
    t = tTemp + matrix(Rq, k,1,s2)
    pk = (A, t)
    sk = (A, t, s1, s2)

```

```

return pk, sk

# assinatura de uma mensagem M
def sign(sk, M):
    A, t, s1, s2 = sk
    y = sample(y1-1, 1)
    Ay = matrix(A) * matrix(Rq, 1, 1, y)
    w1 = []
    for i in range(2):
        lista = Ay[i]
        for j in lista:
            for v in j.list():
                w1.append(int(highBits(int(Zx(v)), int(2*y2))))
    bt = pickle.dumps(w1) + M
    c = Rq(H(bt))
    z = matrix(Rq,1,1,y) + c*matrix(Rq, 1, 1, s1)
    lb = []
    Ay_cs2 = Ay - c*matrix(Rq, k, 1, s2)
    for i in range(2):
        lista = Ay_cs2[i]
        for j in lista:
            for v in j.list():
                lb.append(int(lowBits(int(Zx(v)), int(2*y2))))
    while infinity_norm(z) >= (y1-beta) or max(bt) >= (y2-beta):
        y = sample(y1-1, 1)
        Ay = matrix(A) * matrix(Rq, 1, 1, y)
        w1 = []
        for i in range(2):
            lista = Ay[i]
            for j in lista:
                for v in j.list():
                    w1.append(int(highBits(int(Zx(v)), int(2*y2))))
        bt = pickle.dumps(w1) + M
        c = Rq(H(bt))
        z = matrix(Rq,1,1,y) + c*matrix(Rq, 1, 1, s1)
        lb = []
        Ay_cs2 = Ay - c*matrix(Rq, k, 1, s2)
        for i in range(2):
            lista = Ay_cs2[i]
            for j in lista:
                for v in j.list():
                    lb.append(int(lowBits(int(Zx(v)), int(2*y2))))
    return z, c

# verificacao da assinatura
def verify(pk, M, sign):

```

```

z, c = sign
A, t = pk
Az = A * z
ct = c * t
Azct = Az - ct
w1 = []
for i in range(2):
    lista = Azct[i]
    for j in lista:
        for v in j.list():
            w1.append(int(highBits(int(Zx(v)), int(2*y2))))
bt = pickle.dumps(w1) + M
c_novo = Rq(H(bt))
if infinity_norm(z) < (y1-beta) and c==c_novo:
    print("Valid signature")
else:
    print("Invalid signature")

```

```

[3]: pk, sk = gen()
M= os.urandom(32)
signS = sign(sk,M)
verify(pk, M, signS)

```

Valid signature