Estruturas Criptográficas 2022/23 — TP4. Problema 1

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

May 30, 2023

Este problema consiste em implementar a técnica **DILITHIUM** que é um esquema de assinatura digital presente no concurso NIST PQC e usa o esquema LWE básico como ponto de partida. Nesta implementação foi utilizada a abordagem básica do problema, seguinda os passos presentes neste artigo.

1 RESOLUÇÃO DO PROBLEMA

Na classe abaixo é implementado o algoritmo DILITHIUM que irá gerar uma assinatura utilizando uma chave privada, sendo a chave pública utilizada para verificar a autenbticidade da assinatura. Assim sendo, será necessário implementar 3 funcionalidades principais:

Geração do par de chaves: A função `key_gen` tem como objetivo gerar o par de chaves a ser utilizado para a assinatura da mensagem e para a verificação da assinatura. Para isso começamos por gerar a matriz A de polinómios em $R_q^{k \times l}$, usando o *MatrixSpace()*.

Posteriormente são gerados os vetores $\mathbf{s}_1$ em S_η^l e $\mathbf{s}_2$ em S_η^k, de forma indêntica à criação da matriz A mas direcionada para vetores. Cada coeficiente destes vetores é um elemento de R_q com coeficientes pequenos de tamanho máximo η.

De seguida geramos o vetor t através da expressão $t = A * \mathbf{s}_1 + \mathbf{s}_2$.

No final a chave pública é dada por $p\_key = (A, t)$, já a chave privada será dada por $s\_key = (A, t, \mathbf{s}_1, \mathbf{s}_2)$.

Geração da assinatura: A função `sign` tem como objetivo assinar uma mensagem a ser enviada, devolvendo a assinatura gerada. Para isso, utilizamos a chave privada s_key e a mensagem em bytes $message$. Para isso, inicialmente é gerado um vetor $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^l$, um vetor de polinómios com coeficientes menores ou iguais a γ_1.

Depois é calculado $w := Ay$ e obtido $\mathbf{w}_1 := high\_bits(w, 2\gamma_2)$, que são os *bits* de "ordem maior" dos coeficientes do vetor w.

Posteriormente é gerado $c := H(message||\mathbf{w}_1)$, em que H é instanciado como SHAKE-256 e calcular a assinatura com $z := y + c\mathbf{s}_1$.

Neste caso, se z fosse retornado, o esquema de assinatura seria inseguro pois a chave privada seria revelada. Para evitar a dependência da chave privada, neste esquema é usado *rejection sampling*, para isso são feitas 2 verificações:

1. Se algum coeficiente de z for maior que $\gamma_1 - \beta$, z é rejeitada e recomeçamos o procedimento de assinatura.

2. Ainda, se algum coeficiente dos *bits* de "baixa ordem" de Az - ct for maior que $\gamma_2 - \beta$, é também recomeçado o procedimento de assinatura, sendo que a potencial assinatura z é rejeitada.

A primeira verificação é necessária para a segurança do esquema de assinatura, mas a segunda é necessária tanto para segurança como para a sua correção.

Caso tudo corra bem, e as verificações acima não se verificarem, é devolvida a assinatura $\sigma = (z, c)$.

Verificação da assinatura: A função `verify` tem como objetivo verificar a autenticidade da assinatura σ recebida como parâmetro quando associada à mensagem *message* utilizando para isso a a chave pública p_key. Começamos então por calcular $\mathbf{w'}_1 := high\_bits(Az - ct)$, que corresponde aos *bits* de maior ordem do vetor resultante da operação $Az - ct$.

A assinatura é válida, se todos os coeficientes de z forem menores que $\gamma_1 - \beta$ e se $c*$ corresponder à hash (função H instanciada como SHAKE-256) da concatenação de *message* com $\mathbf{w'}_1$.

Nota: Podemos verificar que o cálculo de $\mathbf{w'}_1$ é bastante semelhante ao cálculo realizado na função de assinatura. Para perceber como esta verificação funciona, é precisor perceber porque é que $high\_bits(Ay, 2\gamma_2) = high\_bits(Az - ct, 2\gamma_2)$. A primeira coisa a reparar é que $Az - ct = Ay - c\mathbf{s}_2$ e, por isso, na realidade o que precisamos de perceber é que $high\_bits(Ay, 2\gamma_2) = high\_bits(Ay - c\mathbf{s}_2, 2\gamma_2)$.

A razão disto é o facto de uma assinatura válida ter sempre $||low\_bits(Ayc\mathbf{s}_2, 2\gamma_2)||_\infty < \gamma_2\beta$. Como sabemos que os coeficientes de $c\mathbf{s}_2$ são menores que β, sabemos também que adicionar $c\mathbf{s}_2$ a Ay não é o suficiente para aumentar qualquer coeficiente de "baixa ordem" de maneira a que tenha magnitude de pelo menos γ_2.

[1]:
```
import hashlib

class DILITHIUM:

    # Parâmetros da técnica DILITHIUM - NIST level 5 - 5+
    def __init__(self):
        self.n = 256
        self.d = 13
        #2^23  2^13 + 1
        self.q = 8380417
        self.k = 8
        self.l = 7
        self.eta = 2
        self.tau = 60
        self.beta = 120
        self.gama_1 = 2^19
        self.gama_2 = (self.q)-1/32
        self.omega = 75

        # Anéis
        Zx.<x> = ZZ[]
        Zq.<z> = PolynomialRing(GF(self.q))
        self.Rq = QuotientRing(Zq,z^self.n+1)
```

```
        self.R = QuotientRing(Zx, x^self.n+1)

        # Espaço matrix
        self.Mr  = MatrixSpace(self.Rq,self.k,self.l)

    # Algoritmo de geração de chaves
    def key_gen(self):
        # Matriz A
        A = self.gen_a()

        # Vetores s1 e s1
        s1 = self.gen_s(self.eta, self.l)
        s2 = self.gen_s(self.eta, self.k)

        t = A*s1 + s2

        p_key = (A,t)
        s_key = (A,t,s1,s2)

        return p_key, s_key

    # Matriz A em Rq
    def gen_a(self):
        K = []
        for i in range(self.k*self.l):
            K.append(self.Rq.random_element())
        A = self.Mr(K)
        return A

    # Vetores S em Rq com o coeficiente até 'limit' e tamanho 'size'
    def gen_s(self, limit, size):
        vetor = MatrixSpace(self.Rq,size,1)
        K = []
        for i in range(size):
            poli = []
            for j in range(self.n):
                poli.append(randint(1,limit))
            K.append(self.Rq(poli))
        S = vetor(K)
        return S

    def sign(self, s_key, message):
        A, t, s1, s2 = s_key

        z = 0
        while(z==0):
            # Vetor y
```

```python
            y = self.gen_s(int(self.gama_1-1) , self.l)

            # w := Ay
            w = A * y

            # w1 := HighBits(w, 2*γ2)
            w1 = self.hb_poli(w, 2*self.gama_2)

            # c  Bτ := H(M || w1)
            c = self.hash(message.encode(), str(w1).encode())
            cq = self.Rq(c)

            # z := y + cs1
            z = y + cq*s1

            if self.norma_inf_vet(z)[0] >= self.gama_1 - self.beta or self.norma_inf_matriz(self.lb_poli(A*y-cq*s2,2*self.gama_2)) >= self.gama_2-self.beta:
                z=0
            else:
                sigma = (z,c)
                return sigma

    # Extrai os “higher-order” bits do decompose
    def high_bits(self, r, alpha):
        (r1,_) = self.decompose(r, alpha)
        return r1

    # Extrai os “lower-order” bits do decompose
    def low_bits(self, r, alpha):
        (_,r0) = self.decompose(r, alpha)
        return r0

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

    def hb_poli(self, poli,alpha):
        k = poli.list()
        for i in range(len(k)):
            h = k[i]
```

```python
            h = h.list()
            for j in range(len(h)):
                h[j]=self.high_bits(int(h[j]), alpha)
            k[i]=h
        return k

    def lb_poli(self,poli,alpha):
        k = poli.list()
        for i in range(len(k)):
            h = k[i]
            h = h.list()
            for j in range(len(h)):
                h[j] = self.low_bits(int(h[j]),alpha)
            k[i] = h
        return k

    # Converte de Bytes para bits
    def access_bit(self, data, num):
        base = int(num // 8)
        shift = int(num % 8)
        return (data[base] & (1<<shift)) >> shift

    # Implementação da função "Hashing to a Ball"
    def sample_in_ball(self,r):
        sl = [self.access_bit(r[:8],i) for i in range(len(r[:8])*8)]
        # Inciar a partir do index 8
        k = 8
        c = [0] * 256

        for i in range (256-self.tau, 256):
            while (int(r[k])>i):
                k +=1

            j = int(r[k])
            k += 1
            s = int(sl[i-196])

            c[i] = c[j]
            c[j] = (-1)^(s)
        return c

    def shake(self,a,b):
        shake = hashlib.shake_256()
        shake.update(a)
        shake.update(b)
        s = shake.digest(int(256))
        return s
```

```
def hash(self,a,b):
    r = self.shake(a,b)
    c = self.sample_in_ball(r)
    return c

def norma_inf(self,pol):
    J = pol.list()
    for i in range(len(J)):
        k = J[i]
        K = k.list()
        for j in range(len(K)):
            K[j] = abs(int(K[j]))
        J[i] = K
    L = []
    for i in range(len(J)):
        L.append(max(J[i]))
    return max(L)

def norma_inf_vet(self,vector):
    for i in range(vector.nrows()):
        norm = self.norma_inf(vector[i])
        vector[i] = norm
    return max(vector)


def norma_inf_matriz(self,matrix):
    L = []
    for i in range(len(matrix)):
        k = matrix[i]
        for j in range(len(k)):
            if k[j] < 0:
                k[j] = abs(k[j])
            L.append(max(k))
    for i in range(len(L)):
        J = []
        J.append(max(L))
    return J[0]

# Verifica a assinatura na mensagem utilizando a p_key
def verify(self,p_key, message, sigma):
    A,t = p_key
    z,c = sigma

    cq = self.Rq(c)

    w1 = self.hb_poli(A*z - cq*t, 2*self.gama_2)
```

```python
        u = str(w1).encode()
        k = message.encode()
        c_ = self.hash(k,u)

        return self.norma_inf_vet(z)[0] < self.gama_1 - self.beta and c_ == c
```

2 Resultados

[2]:
```python
dilithium = DILITHIUM()

message = 'This is the message'

wrong_message = 'This message is wrong'

p_key,s_key = dilithium.key_gen()

sigma = dilithium.sign(s_key, message)

result = dilithium.verify(p_key, message, sigma)

print("Verifying the correct message:")
if result:
    print("Valid signature.")
else:
    print("Invalid signature.")

wrong_result = dilithium.verify(p_key, wrong_message, sigma)

print("Verifying the incorrect message:")
if wrong_result:
    print("Valid signature.")
else:
    print("Invalid signature.")
```

```
Verifying the correct message:
Valid signature.
Verifying the incorrect message:
Invalid signature.
```