Dilithium

May 30, 2023

1 TP4

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 1.

Neste trabalho pretende-se implementar em Sagemath de algumas dos candidatos a "standartização" ao concurso NIST Post-Quantum Cryptography na categoria de esquemas de assinatura digital. Ver também a directoria com a documentação. Construa

1. Um protótipo Sagemath do algoritmo Dilithium ,

2 Dilithium

[1]:
```
# imports
from sage.all import *
from cryptography.hazmat.primitives import hashes
```

2.0.1 INFO

Este algoritmo baseia-se em três passos principais: - Geração das chaves (pública e privada) quando o algoritmo é instanciado - Função **assinar()**: procedimentos para assinatura - Função **verificar()**: procedimentos para a verificação efectiva de uma assinatura

Além disso, um dos objectivos deste algoritmo é ser modular e parametrizável, pelo que foram implementados vários modos de instanciação com diferentes níveis de segurança nos parâmetros propostos.

As seguintes classes são passadas como argumentos para o construtor da classe *Dilithium*

[2]:
```
class Weak:
    k = 3
    l = 2
    eta = 7
    beta = 375
    omega = 64
```

```
class Medium:
    k = 4
    l = 3
    eta = 6
    beta = 325
    omega = 80

class Recommended:
    k = 5
    l = 4
    eta = 5
    beta = 275
    omega = 96

class VeryHigh:
    k = 6
    l = 5
    eta = 3
    beta = 175
    omega = 120
```

2.0.2 Implementação

Geração das chaves: O algoritmo de geração de chaves gera uma matriz A de dimensões kxl, e amostra 2 vetores **s1** e **s2**. Também gera um último parâmetro público $t = A * s1 + s2$. Assim, para mostrar a matriz A e os vetores de polinómios **s1** e **s2**, bastou-nos implementar dois métodos auxiliares, que seguem a especificação nos documentos (nomeadamente, expandA e sample). Uma vez geradas todas estas variáveis, finalmente temos as chaves: **Chabe Pública: (A, t)** e **Chave Privada: (A, t, s1, s2)**.

Assinatura: O algoritmo de assinatura necessita de seguir uma série de passos: - É amostrado y com dimenção igual a $l \times 1$. De seguida, calcula-se os *high_bits* de $A * y$ para *w1* - Obter o hash **H()** a partir de *w1* e da mensagem - Calcular $z = y + c * s1$ - Finalmente, é necessário verificar a condição de assinatura. Caso não seja satisfeita, efetuar novamente o processo

Verificação: Para se verificar a assinatura a partir da chave pública, basta seguir os seguintes passos: - Calcular os *high_bits* de $A * y - c * t$ para *w1* - De seguida, basta confirmar se a condição da assinatura se verifica

Todos estes algoritmos implicam uma série de métodos auxiliares, tal como estão especificados nos documentos oficiais. Deste modo, foram também implementados e comentados de seguida.

[3]:
```
class Dilithium:
    def __init__ (self, params=Recommended):
        # Definição de parametros
        self.n = 256
        self.q = 8380417
```

```
        self.d = 14
        self.weight = 60
        self.gamma1 = 523776 # (q-1) / 16
        self.gamma2 = 261888 # gamma1 / 2
        self.k = params.k
        self.l = params.l
        self.eta = params.eta
        self.beta = params.beta
        self.omega = params.omega

        # Definir os campos
        Zq.<x> = GF(self.q)[]
        self.Rq = Zq.quotient(x^self.n + 1)

        # Geração de chaves
        self.A = self.expandirA()
        self.s1 = self.sample(self.eta, self.l)
        self.s2 = self.sample(self.eta, self.k)
        self.t = self.A * self.s1 + self.s2

        # chave publica : A, t
        # chave privada : s1, s2

    def assinar(self, m):
        # Inicialização da Variável
        z = None
        # caso nenhum 'z' tenha sido gerado
        while z == None:
            # Inicio de geração de 'z'
            y = self.sample(self.gamma1 - 1, self.l)
            #Ay é reutilizado por isso pode-se precalcular
            Ay = self.A * y
            # High bits
            w1 = self.high_bits(Ay, 2 * self.gamma2)
            # Calculo do Hash
            hsh = self.H(b"".join([bytes([ int(i) for i in e]) for e in w1]) +␣
↪m)
            # Calculo do polinómio
            hash_poly = self.Rq(hsh)

            # Calculo do 'z'
            z = y + hash_poly * self.s1

            # Verificar as condições
            if(self.sup_normal(z) >= self.gamma1 - self.beta) and (self.
↪sup_normal([self.low_bits(Ay - hash_poly * self.s2, 2 * self.gamma2)]) >=␣
↪self.gamma2 - self.beta):
```

```
                # Necessário calcular um novo 'z'
                z = None

            return (z,hsh)


    def verificar(self, m, ass):
        # Assinatura
        (z,hsh) = ass

        # Calcular os High Bits
        w1_ = self.high_bits(self.A * z - self.Rq(hsh) * self.t, 2 * self.
↪gamma2)

        # Calcular as condições de verificação
        torf1 = (self.sup_normal(z) < self.gamma1 - self.beta)
        torf2 = (hsh == self.H(b"".join([bytes([ int(i) for i in e]) for e in␣
↪w1_]) + m))

        # torf1 && torf2
        return torf1 and torf2

    def expandirA(self):
        mat = [ self.Rq.random_element() for _ in range(self.k * self.l)]
        return matrix(self.Rq, self.k, self.l, mat)

    def sample(self, coef_max, size):
        def rand_poly():
            return self.Rq( [randint(0,coef_max) for _ in range(self.n)])

        vector = [rand_poly() for _ in range(size)]

        return matrix(self.Rq, size, 1, vector)

    def high_bits(self, r, alfa):
        r1, _ = self.decompose(r, alfa)
        return r1

    def low_bits(self, r, alfa):
        _, r0 = self.decompose(r, alfa)
        return r0

    def decompose(self, r, alfa):

        r0_vector = []
        r1_vector = []
        torf = True
```

```
        for p in r:
            r0_poly = []
            r1_poly = []

            for c in p[0]:
                c = int(mod(c, int(self.q)))
                r0 = int(mod(c, int(alfa)))

                if c - r0 == int(self.q) - int(1):
                    r1 = 0
                    r0 = r0 - 1
                else:
                    r1 = (c - r0) / int(alfa)

                r0_poly.append(r0)
                r1_poly.append(r1)

            if torf:
                torf = False

            r0_vector.append(self.Rq(r0_poly))
            r1_vector.append(self.Rq(r1_poly))

        return (r1_vector, r0_vector)

    def H(self, obj):
        sha3 = hashes.Hash(hashes.SHAKE256(int(60)))
        sha3.update(obj)
        res = [ (-1) ** (b % 2) for b in sha3.finalize()]
        return res + [0] * 196

    def sup_normal(self, v):
        return max([max(p[0]) for p in v])
```

2.0.3 Testes

Foram definidos três diferentes testes para certificar que as assinaturas estão a ser bem geradas. Para tal, instanciou-se duas classes diferentes, com os mesmos parâmetros.

[4]:
```
dilithium1 = Dilithium(params = Recommended)
dilithium2 = Dilithium(params = Recommended)
mensagem = b"Grupo 17, EC 2022/2023"
```

Teste 1: Verificar se o esquema valida corretamente uma assinatura

[5]:
```
# Assinar uma mensagem
ass = dilithium1.assinar(mensagem)
# Verificar a assinatura
print("Teste 1 (True):", dilithium1.verificar(mensagem, ass))
```

```
Teste 1 (True): True
```

Teste 2: Verificar se o esquema reconhece quando os dados assinados são diferentes

[7]:
```
# Assinar uma mensagem
ass = dilithium1.assinar(mensagem)
# Verificar a assinatura
print("Teste 2 (False):", dilithium1.verificar(b"Estruturas Criptograficas",␣
 ↪ass))
```

```
Teste 2 (False): False
```

Teste 3: Verificar se existe relação entre instâncias diferentes

[8]:
```
# Assinar uma mensagem
ass = dilithium1.assinar(mensagem)
# Verificar a assinatura
print("Teste 3 (False):", dilithium2.verificar(mensagem, ass))
```

```
Teste 3 (False): False
```