Dilithium

May 10, 2023

1 Estruturas Criptográficas: Trabalho Prático 4

Grupo 13, constituído por: – Rodrigo Rodrigues, PG50726
– Rui Guilherme Monteiro, PG50739

1.1 Dilithium

Referências: https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
https://eprint.iacr.org/2017/633.pdf

1.1.1 Imports

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

1.1.2 Definir vários modos de instânciação, com diferentes níveis de segurança nos parâmetros propostos.

[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
```

1.1.3 Definir o algoritmo

O algoritmo assenta em três passos principais:

1. Geração das chaves (pública e privada) na instanciação do algoritmo.
2. sign()
3. verify()

[3]:
```
class Dilithium:
    def __init__(self, params=Recommended):
        self.n = 256
        self.q = 8380417
        self.d = 14
        self.weight = 60
        self.gamma1 = 523776    # (self.q-1) / 16
        self.gamma2 = 261888    # self.gamma1 / 2
        self.k = params.k
        self.l = params.l
        self.eta = params.eta
        self.beta = params.beta
        self.omega = params.omega

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

        # Gerar as chaves
        self.A = self.expandA()
        self.s1 = self.generate_random_vector(self.eta, self.l)
        self.s2 = self.generate_random_vector(self.eta, self.k)
        self.t = self.A * self.s1 + self.s2
        # Public Key : A, t
        # Private Key : s1, s2


    def sign(self, m):
        """
```

```
            Assinar uma mensagem m, em bytes.
        """

        z = None
        while z == None:
            # começar o processo de gerar 'z':
            y = self.generate_random_vector(self.gamma1-1, self.l)
            # Ay é reutilizado por isso precalcula-se
            Ay = self.A * y
            # high bits
            w1 = self.get_high_order_bits(self.A * y, 2 * self.gamma2)
            # calcular o hash
            c = self._hash(b"".join([bytes([ int(i) for i in e ]) for e in w1])
↪+ m)
            # calcular o polinómio
            c_poly = self.Rq(c)

            # calcular o 'z'
            z = y + c_poly * self.s1

            # verificar as condições
            if (self.sup_norm(z) >= self.gamma1 - self.beta) and (self.
↪sup_norm([self.get_low_order_bits(Ay-c_poly*self.s2, 2*self.gamma2)]) >=
↪self.gamma2 - self.beta):
                # é necessário calcular novo 'z'
                z = None

        return (z,c)

    def verify(self, m, sig):
        """
            Função de verificação de uma mensagem.

            Parâmetros:
            ----------
            m: mensagem em bytes

            sig: assinatura
        """

        # assinatura
        (z,c) = sig

        # calcular os high bits
        w1_ = self.get_high_order_bits(self.A*z - self.Rq(c)*self.t, 2*self.
↪gamma2)
```

```
        # calcular condições de verificação
        torf1 = (self.sup_norm(z) < self.gamma1-self.beta)
        torf2 = (c == self._hash(b"".join([bytes([ int(i) for i in e ]) for e␣
↪in w1_]) + m))

        return torf1 and torf2

    ############################################ Métodos AUXILIARES␣
↪##############################################


    def expandA(self):
        """
            Mapear uma seed  {0, 1}^256 numa matriz A  Rq^k×l
        """
        mat = [ self.Rq.random_element() for _ in range(self.k*self.l) ]
        return matrix(self.Rq, self.k, self.l, mat)

    def generate_random_vector(self, coef_max, size):
        """
            Gera um vetor aleatório onde cada coeficiente desse vetor é um␣
↪elemento pertencente a Rq.
        """
        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) # vetor representado sob forma de␣
↪matriz p/ permitir ops com a matriz A

    def get_high_order_bits(self, r, alfa):
        """
            Obter os bits de ordem superior.
        """
        r1, _ = self.extract_bits(r,alfa)
        return r1

    def get_low_order_bits(self, r, alfa):
        """
            Obter os bits de ordem inferior.
        """
        _, r0 = self.extract_bits(r,alfa)
        return r0

    def extract_bits(self, r, alfa):
        """
```

```
            Extrai bits de higher-order e lower-order de elementos pertencentes␣
↪a Zq.
        """

        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 _hash(self, obj):
        """
            Função de hash que usa SHAKE256 para construir um array com 256␣
↪elementos de -1 a 0.
        """
        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_norm(self, v):
        """
            Uniform norm / "norma do supremo" / "normal uniforme".
            Ref: https://en.wikipedia.org/wiki/Uniform_norm
        """
        return max([ max(p[0]) for p in v])
```

1.1.4 Testes

Para tal, instanciou-se duas classes diferentes, com os mesmmos parâmetros.

[4]:
```
dilithium = Dilithium(params=Recommended)
dilithium_B = Dilithium(params=Recommended)
```

Teste 1: Verificar se o esquema valida corretamente uma assinatura.

[5]:
```
# Sign
signature = dilithium.sign(b"Mensagem secreta do Grupo 13.")

# Verify
print("Teste 1: ", dilithium.verify(b"Mensagem secreta do Grupo 13.",␣
 ↪signature))
```

```
Teste 1:  True
```

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

[6]:
```
# Sign
signature = dilithium.sign(b"Mensagem secreta do Grupo 13.")

# Verify
print("Teste 2: ", dilithium.verify(b"Enemy attacks tonight.", signature))
```

```
Teste 2:  False
```

Teste 3: Verificar se entre instâncias diferentes não existem relações.

[7]:
```
# Sign
signature = dilithium.sign(b"Mensagem secreta do Grupo 13.")

# Verify
print("Teste 3: ", dilithium_B.verify(b"Mensagem secreta do Grupo 13.
 ↪",signature))
```

```
Teste 3:  False
```