Trabalho Prático 3

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

CRYSTALS-Dilithium

> Dilithium is a digital signature scheme that is strongly secure under chosen message attacks based on the hardness of lattice problems over module lattices. The security notion means that an adversary having access to a signing oracle cannot produce a signature of a message whose signature he hasn't yet seen, nor produce a different signature of a message that he already saw signed.
>
> https://pq-crystals.org/dilithium/

Este algoritmo, Dilithium, foi desenhado com o objetivo de satisfazer os seguintes critérios:

- *Simple to implement securely*
- *Be conservative with parameters*
- *Minimize the size of public key + signature*
- *Be modular – easy to vary security*

Deste modo, toda a nossa implementação procura seguir dois dos documentos de CRYSTALS-Dilithium:
https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf
https://eprint.iacr.org/2017/633.pdf

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

Este algoritmo assenta em três passos principais:

- Geração das chaves (pública e privada) na instanciação do algoritmo.
- Função **sign()**: tratamento para efetivamente efetuar uma assinatura.
- Função **verify()**: tratamento para efetivamente verificar uma assinatura.

Além disso, este algoritmo tem como um dos objetivos ser modular e parameterizável, pelo que, então, implementou-se vários modos de instânciação, com os diferentes níveis de segurança nos parâmetros propostos.

As seguintes classes são passadas como argumento ao construtor do Dilithium.

In [ ]:
```
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
```

Implementação

Geração das chaves:

O algoritmo de geração de chaves gera uma **matriz A** de dimensões **k x l**, e amostra 2 vetores **s1** e **s2**. Também, gera um último parâmetro público **t = A*s1 + s2**.

Assim, para amostrar 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: **Public Key: (A, t)** e **Private Key: (A, t, s1, s2)**.

Assinatura:

O algoritmo de assinatura necessita de seguir uma série de passos:

- É amostrado **y** com dimenção igual a **l×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:

- Calcula-se 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.

In [ ]:
```
class Dilithium:
    def __init__(self, params=Recommended):
        # Define Parameters
        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)

        # Generate Keys
        self.A = self.expandA()
        self.s1 = self.sample(self.eta, self.l)
        self.s2 = self.sample(self.eta, self.k)
        self.t = self.A * self.s1 + self.s2
        # Public Key : A, t
        # Private Key : s1, s2

    # função de assinatura de uma mensagem
    # m: mensagem em bytes
    def sign(self, m):
        # inicialização da variável
        z = None
        # se nenhum 'z' foi gerado
        while z == None:
            # começar o processo de gerar 'z':
            y = self.sample(self.gamma1-1, self.l)
            # Ay é reutilizado por isso precalcula-se
            Ay = self.A * y
            # high bits
            w1 = self.high_bits(self.A * y, 2 * self.gamma2)
            # calcular o hash
            c = self.H(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.low_bits(Ay-c_poly*self.s2, 2*self.gamma2)]) >= self.gamma2 - self.beta):
                # é necessário calcular novo 'z'
                z = None

        return (z,c)

    # função de verificação de uma mensagem
    # m: mensagem em bytes
    # sig: assinatura
```

```
    def verify(self, m, sig):
        # assinatura
        (z,c) = sig
        # calcular os high bits
        w1_ = self.high_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.H(b"".join([bytes([ int(i) for i in e ]) for e in w1_]) + m))

        # torf1 && torf2
        return torf1 and torf2

    ################################ Funções Auxiliares ################################

    # Mapear uma seed ∈ {0, 1}^256 numa matriz A ∈ Rq^k×l
    def expandA(self):
        # Na submissão original assume-se ϱ como uma seed uniforme para amostrar aleatoriamente.
        # Neste caso considera-se que `random_element` tem o valor equivalente da seed internamente.
        mat = [ self.Rq.random_element() for _ in range(self.k*self.l) ]
        return matrix(self.Rq, self.k, self.l, mat)

    # gera um vetor aleatório onde cada coeficiente desse vetor é um elemento pertencente a Rq
    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) ]

        # Vectores sao representados sob a forma de matrizes para permitir as operações com a matriz A
        return matrix(self.Rq,size,1,vector)

    # recupera os bits de ordem superior
    def high_bits(self, r, alfa):
        r1, _ = self.decompose(r,alfa)
        return r1

    # recupera os bits de ordem inferior
    def low_bits(self, r, alfa):
        _, r0 = self.decompose(r,alfa)
        return r0

    # extrai bits de higher-order e lower-order de elementos pertencentes a Zq
    def decompose(self, r, alfa):
        # Nota: Na submissão original é assumido que as operações no decompose são aplicadas a cada coeficiente.
        # r1 r0
        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))
        # não se realiza mais operações sobre matrizes, então retornar vectores
        return (r1_vector, r0_vector)

    # função de hash que recorre a SHAKE256 de modo a construir um array com 256 elementos de -1 a 0
    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

    # normal uniforme
    # https://en.wikipedia.org/wiki/Uniform_norm
    def sup_norm(self, v):
        return max([ max(p[0]) for p in v])
```

Testes

Nesta secção iremos efetuar três diferentes testes para certificar que as assinaturas estão a ser bem geradas.

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

In [ ]:
```
# instanciar a classe (Parâmetros = Recommended)
dilithium = Dilithium(params=Recommended)
dilithium_other = Dilithium(params=Recommended)
```

Teste 1: Neste cenário, verifica-se se o esquema valida corretamente uma assinatura.

In [ ]:
```
# Assinar uma mensagem
sig = dilithium.sign(b"Grupo 17, EC 2021/2022")
# Verificar a assinatura
print("Test 1 (Must be True):",dilithium.verify(b"Grupo 17, EC 2021/2022", sig))
```

```
Test 1 (Must be True): True
```

Teste 2: Neste cenário, verifica-se se o esquema reconhece quando os dados assinados são diferentes.

In [ ]:
```
# Assinar uma mensagem
sig = dilithium.sign(b"Grupo 10, EC 2021/2022")
# Verificar a assinatura
print("Test 2 (Must be False):",dilithium.verify(b"To be or not to be", sig))
```

```
Test 2 (Must be False): False
```

Teste 3: Neste cenário, verifica-se se entre instâncias diferentes não existem relações.

In [ ]:
```
# Assinar uma mensagem
sig = dilithium.sign(b"Grupo 10, EC 2021/2022")
# Verificar a assinatura
print("Test 3 (Must be False):",dilithium_other.verify(b"Grupo 10, EC 2021/2022",sig))
```

```
Test 3 (Must be False): False
```