Trabalho prático 3

Grupo 5:

- Duarte Oliveira \<pg47157>
- Melânia Pereira \<pg47520>

Rainbow

Criação de um protótipo em Sagemath do algoritmo Rainbow, candidato ao concurso NIST Post-Quantum Cryptography na categoria de esquemas de assinatura digital.

Inicialização

In [550…

```
import hashlib
```

In [551…

```
class RAIN:
    #Geração dos parâmetros (Security Category I)
    def __init__(self):
        self.q = 16
        self.u = 2
        self.v1 = 36
        self.o1 = 32
        self.o2 = 32
        self.v2 = self.v1+1+self.o1

        self.m = self.o1 + self.o2
        self.n = self.m + self.v1

        self.F = GF(self.q)
        self.FF = PolynomialRing(self.F,names=['x'+str(i) for i in range(1,self.n+1)])
```

Funções auxiliares

Definimos as funções *rainbowMap()*, *H()* e *invF()*

- *rainbowMap()* é a função que gera um mapa central de acordo com os parâmetros do algoritmo com coeficientes escolhidos aleatorimente em F_q.
- *H()* é uma função de hash que usa o SHAKE-256.
- *InvF()* é a função de inversão do mapa gerado pela *rainbowMap()*.

In [552…

```
class RAIN(RAIN):

    #Definição de funções auxiliares
    def rainbowMap(self):
        v2 = self.v1+self.o1
        v3 = self.n

        V = [range(1, self.v1+1), range(1, v2+1)]
        O = [range(self.v1+1, v2+1), range(v2+1, v3+1)]

        f=[]
        for k in range(self.v1+1, self.n+1):
            try:
                pol = 0
                var = self.FF.gens()
                l = 1
                if k in O[0]:
                    l = 0

                for i in V[l]:
                    for j in V[l]:
                        pol += self.F.random_element()*var[i-1]*var[j-1]
                        pol += self.F.random_element()*var[i-1]
                    for j in O[l]:
                        pol += self.F.random_element()*var[i-1]*var[j-1]
                        pol += self.F.random_element()*var[j-1]

                pol += self.F.random_element()

                f.append(self.FF(pol))
            except Exception as e:
                print('e: ', l, k,j)

        return f


    def H(self, d):
        dig = hashlib.shake_256(d.encode()).digest(int(32))

        h = []
        lF = self.F.list()
        for i in dig:
            h.append(lF[int(format(i, '08b')[:4], 2)])
            h.append(lF[int(format(i, '08b')[4:], 2)])
        return h


    def invF(self, Fm, x):
        t = False
        while not t:

            y = [self.F.random_element() for _ in range(self.v1)]
            vars = self.FF.gens()
            aux = {}
            for i in range(self.v1):
                aux[vars[i]] = y[i]

            Fm_ = []
            for f in Fm:
                Fm_.append(f.subs(aux))


            lin_sys = []
            for i, p in enumerate(Fm_[:32]):
                lin_sys.append(p-self.FF(x[i]))

            GG = PolynomialRing(self.F,names=vars[36:68])

            J = GG.ideal(lin_sys)
            if J.dimension() == 0:
                variety =  J.variety()
                if len(variety)!= 0:
                    y2 = [v for v in  variety[0].values()]
                    y2.reverse()
                    aux = {}
                    for i, v in enumerate(y2):
                        aux[vars[i+self.v1]] = v

                    lin_sys = []
                    for i, p in enumerate(Fm_[32:]):
                        lin_sys.append(p.subs(aux)-self.FF(x[i+32]))

                    GG = PolynomialRing(self.F,names=vars[68:])
                    y += y2
                    y2=[]

                    J = GG.ideal(lin_sys)
                    if J.dimension() == 0:
                        variety =  J.variety()
                        if len(variety)!= 0:
                            for s in variety[0].values():
                                y2.append(s)
                            t=True
                            y2.reverse()
        return y+y2
```

Geração de chaves

Para a chave privada, é necessário gerar dois *affine maps* $S : F^m \to F^m$ e $T : F^n \to F^n$ e ainda um *central map* quadrático $F : F^n \to F^m$. Assim, a chave privada é composta pelas matrizes e vetores que constroem esses mapas.

A chave pública consiste na composição de S com F e com T.

In [553…

```
class RAIN(RAIN):

    #Algoritmo de geração de chaves
    def keygen(self):
        MQ = self.FF^self.m
        Fn = self.F^self.n
        Fm = self.F^self.m

        Ms = matrix.random(self.F, self.m, self.m)

        while not Ms.is_invertible():
            Ms = matrix.random(self.F, self.m, self.m)

        cs = matrix(self.m, 1, Fm.random_element())

        def S(x):
            return Ms*x + cs

        invS = Ms.inverse()

        Mt = matrix.random(self.F, self.n, self.n)

        while not Mt.is_invertible():
            Mt = matrix.random(self.F, self.n, self.n)

        ct = matrix(self.n, 1, Fn.random_element())

        def T(x):
            return vector(Mt*x + ct)

        invT = Mt.inverse()

        polf = self.rainbowMap()

        def f(x):
            r=[]
            for f in polf:
                r.append(f(*x))
            return matrix(self.m, 1, r)

        comp = compose(S,compose(f,T))
        p = comp(matrix(self.n, 1, self.FF.gens()))

        sk = (invS,cs,polf,invT,ct)
        pk = p

        return (sk,pk)
```

Assinatura

Para assinar uma qualquer mensagem *d*, começamos por gerar o seu *hash*, através da função de *hashing* SHAKE-256.

Depois é calculado **x** = $S^{-1}(h)$ e uma pré imagem desse **x** sob o *central map* **F** através da função *InvF()* definida acima.

Finalmente, calculamos a assinatura **z** = $T^{-1}(y)$.

In [554…

```
class RAIN(RAIN):

    #Algoritmo de assinatura
    def sign(self,sk,d):
        (invS,cs,f,invT,ct) = sk

        h = matrix(self.m, 1, self.H(d))

        x = invS * (h - cs)

        y = matrix(self.n, 1, self.InvF(f, x))

        z = invT * (y - ct)

        return z
```

Verificação

Dada uma mensagem ou documento **d** e uma assinatura **z**, começamos por calcular a *hash* de **d** tal como foi feito para a geração da assinatura.

Depois, calculamos **h'** através da chave pública, que é a composição das funções geradas na geração das chaves, à qual passamos a assinatura **z**.

Se **h** for igual a **h'**, então a assinatura é verificada.

In [555…

```
class RAIN(RAIN):

    #Algoritmo de verificação
    def ver(self,d,z,pk):
        h = matrix(self.m, 1, self.H(d))

        h_ = pk(z)

        return h==h_
```

Teste

In [556…

```
rain = RAIN()
m = 'hello'

(sk,pk) = rain.keygen()
print('keys generated')
```

```
keys generated
```

In [559…

```
z = rain.sign(sk,m)
print('signed')

print('verified?')
rain.ver(m,z.list(),pk)
```

```
signed
verified?
```

Out[559…

```
True
```