TP3-Problema1-Rainbow

May 30, 2022

1 TRABALHO PRÁTICO 3- GRUPO 14

1.1 Problema 1 - Rainbow

Este problema pretende implementar algumas das candidaturas ao concurso NIST Post-Quantum Cryptography na categoria de esquemas de assinatura digital. Neste caso, o objetivo será criar protótipos em sagemath para o algoritmo Rainbow. Nesta implementação do Rainbow foram utilizados os passos apresentados no documento Rainbow fornecido pelo docente.

IMPORTS

[1]:
```
import sys
import hashlib
sys.setrecursionlimit(10000)
```

1.2 Resolução do Problema

Neste problema era então proposta a implementação do algoritmo de assinatura digital Rainbow.

O grupo decidiu instanciar a classe desenvolvida e optar pela categoria de segurança nível I e II defendida pelo NIST no que concerne a valores de parâmetros e desenvolvimento de função de hashing que são distintas de cada categoria.

Antes de avançar para a explicação da abordagem, é de notar que esta implementação tem um tempo de execução considerável tornando a execução desenvolvida pouco otimizada. O grupo identificou assim que nas fases de gerar o mapa central **F**, e na composição **P** o tempo necessário para executar essas operações era bastante demorado tornando a execução pouco otimizada.

De seguida, apresentamos as funções desenvolvidas bem como uma breve descrição sobre o papel de cada uma na nossa implementação:

Rainbowmap - A função `rainbowmap` é a responsável pelo desenvolvimento do mapa rainbow central. A implementação é baseada num sistema Oil-Vinegar e retorna um mapa rainbow de acordo com os parametros instanciados na classe(q, v1, o1, o2).

InvF - A função `InvF` é responsável pela inversão do mapa central F. A função Gauss retorna um valor binário bool {TRUE, FALSE} indicando se o sistema linear dado é possível e, se for, uma solução aleatória do sistema. Por fim, é devolvido um vector y que pertence a Fn que verifique a igualdade F(y) = x, sendo x um vector passado como input pertencendo a Fm.

key__gen - A função `key_gen` recebe alguns parâmetros instanciados na classe e tem o objetivo de devolver a chave secreta + pública. No caso do algoritmo de assinatura digital Rainbow, a chave secreta é composta pelo tuplo (InvS, cs, polyF, InvT, ct) que consiste em dois "affine maps" invertíveis **S** e **T** e no mapa central quadrático **F**. Por outro lado, a chave pública é obtida por um mapa composto **P**, $P \leftarrow S \circ F \circ T$ e portanto consiste num polinómio quadrático no anel F.

sign - Esta função descreve o processo de geração de uma assinatura no algoritmo Rainbow. Recebe como argumentos o tuplo da chave privada **sk** e um documento **d** que será assinado com a chave. Por fim, é retornada uma assinatura **z** tal que $P(z) = H(d)$ para futura verificação.

verify - A função `verify` tem o papel de verificar a autenticidade de uma assinatura Rainbow. O processo é bastante simples, sendo apenas necessário a hash do documento recebido como argumento com a assinatura recebida. Basicamente, nesta função é verificada a validade da operação $P(z) = H(d)$. Em caso de output verdadeiro(true), a assinatura é autêntica.

H(sha256) - A função de hashing é baseada no sha256 conforme a submissão do Rainbow defende para as categorias de segurança I e II. Esta função só está preparada para devolver vetores com 64 elementos, e por isso teria de ser revista para outros niveis de segurança. Nesta função são selecionados, sucessivamente, 4 bits do digest do SHA-256 e cada um deles representa um elemento de F.

[13]:
```
class Rainbow:
    def __init__(self,q,v1,o1,o2):
        #parametros para categoria de segurança NIST I&II
        #F=GF(16), (v1, o1, o2) = (36,32,32)
        self.q = 16
        self.u = 2
        self.v1 = v1
        self.o1 = o1
        self.o2 = o2
        self.v2 = v1+1+o1

        self.m = o1 + o2
        self.n = self.m + v1
        self.F = GF(self.q)
        self.FF = PolynomialRing(self.F,names=['x'+str(i) for i in range␣
↪(1,self.n+1)])

    # Unbalanced Oil-Vinegar polynomials??
    #funçao: desenvolvimento do mapa central, mapa gerado através de elementos␣
↪aleatorios em F
    #output: retorna um mapa rainbow de acordo com os parametros instanciados␣
↪na classe(q, v1, o1, o2)
    def rainbowmap(self,F,FF,v1,o1,o2,n):
        v2 = v1 + o1
        v3 = n

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

```
    _map = []

    for k in range(v1+1,n+1):
        try:
            poly = 0
            var = FF.gens()
            l = 1
            if k in O[0]:
                l = 0
            for i in V[l]:
                for j in V[l]:
                    poly += F.random_element()*var[i-1]*var[j-1]
                    poly += F.random_element()*var[i-1]
                for j in O[l]:
                    poly += F.random_element()*var[i-1]*var[j-1]
                    poly += F.random_element()*var[j-1]
            poly += F.random_element()
            _map.append(FF(poly))

        except Exception as e:
            print('Something went wrong: ', e)
    return _map


#funçao: inversão do mapa central do rainbow
#inputs: Rainbow central mapF= (f(v1+1), . . . , f(n)), vector x   Fm
def invF(self,Fm,x):
    _bool = False
    while not _bool:

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

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

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

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

```
            j = GG.ideal(linear_sys)
            if j.dimension() == 0:
                variety = j.variety()
                if len(variety) != 0:
                    y1 = [v for v in variety[0].values()]
                    y1.reverse()

                    aux = {}
                    for i,v in enumerate(y1):
                        aux[var[i+self.v1]] = v

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

                    GG = PolynomialRing(self.F, names=var[68:])
                    y += y1
                    y1 = []

                    j = GG.ideal(linear_sys)
                    if j.dimension() == 0:
                        variety = j.variety()
                        if len(variety) != 0:
                            for i in variety[0].values():
                                y1.append(i)
                            _bool=True
                        y1.reverse()
            return y+y1


    #funçao: geração do par de chaves que retorna o tuplo da chave secreta␣
↪(InvS, cs, polyF, InvT, ct) + chave pública (P)
    def key_gen(self):
        #parâmetros necessários da classe para fazer a geração do par de chaves
        m = self.m
        n = self.n
        F = self.F
        FF = self.FF
        MQ = FF^m
        Vn = F^n
        Vm = F^m

        #gerar o mapa S
        Ms = matrix(F,m,m, [F.random_element() for _ in range(m*m)])
        while not Ms.is_invertible():
            Ms = matrix(F,m,m, [F.random_element() for _ in range(m*m)])
        cs = matrix(m,1,Vm.random_element())
```

```
        #S ← Aff (MS, cS)
        def S(x): return Ms*x + cs
        InvS = Ms.inverse()
        print('S map generated')

        #gerar o mapa T
        Mt = matrix(F,n,n, [F.random_element() for _ in range(n*n)])
        while not Mt.is_invertible():
            Mt = matrix(F,n,n, [F.random_element() for _ in range(n*n)])
        ct = matrix(n,1,Vn.random_element())
        #T ←Aff(MT, cT)
        def T(x): return vector(Mt*x + ct)
        InvT = Mt.inverse()
        print('T map generated')

        #gerar F a partir do rainbowmap
        polyF = self.rainbowmap(F,FF, self.v1,self.o1,self.o2,n)
        print('F map generated')

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

        #gerar P, public key que é o compose dos mapas S T F
        comp = compose(S,compose(f,T))
        p = comp(matrix(n,1,FF.gens()))
        print('P generated')

        #tuplo de chave privada + chave pública
        sk = (InvS, cs, polyF, InvT, ct)
        pk = p
        return sk,pk

    #funçao: processo de geração de uma assinatura com o sistema rainbow
    # input: chave privada + o documento para assinar
    def sign(self,sk,d):
        #sk = (InvS, cS,F, InvT, cT)
        h = matrix(self.m,1,self.H(d))
        #y = InvF(F, x) sendo x = InvS * (h-cS)
        y = matrix(self.n,1,self.invF(sk[2], sk[0] * (h - sk[1])))

        #return z -> InvT·(y-cT)
        return sk[3]*(y - sk[4])

    #funçao: verificar a assinatura rainbow
```

```
    #inputs: documento + uma signature
    def verify(self,pk,z,d):
        #h← H(d)
        h = matrix(self.m,1,self.H(d))
        h1 = pk(z)

        if h == h1:
            return true
        else:
            return false

    #funçao: funçao de hash utilizando sha256
    def H(self,d):
        digest = hashlib.sha256(d).digest()

        h = []
        listF = self.F.list()
        for i in digest:
            #4 bits to each element of F
            h.append(listF[int(format(i,'08b')[:4],2)])
            h.append(listF[int(format(i,'08b')[4:],2)])
        return h
```

1.2.1 Cenário de Teste

De seguida, apresentamos um cenário de teste para a implementação do Rainbow desenvolvida, onde instanciamos a classe rainbow com os parâmetros de segurança do NIST para as categorias I e II, geramos o par de chaves, criamos um documento **d** para fazer a sua assinatura e de seguida verificámos a integridade dessa assinatura através do método `verify` da classe Rainbow.

[11]:
```
rainbow = Rainbow(16, 36, 32, 32)
sk, pk = rainbow.key_gen()
#documento to sign
d = b'235555'
z = rainbow.sign(sk, d)
#verify
rainbow.verify(pk, z.list(), d)
```

```
S map generated
T map generated
F map generated
P generated
```

[11]: True

[ ]: