Dilithium

June 23, 2020

1 Dilithium

1.1 Parâmetros

Neste esquema de assinaturas: n,q e h são fixos. Os restantes parâmetros são recomendados.

- **n =** 256
- **q =** 2^23 - 2^13 + 1
- **h =** 60
- **r =** 1753 (r é uma raiz primitiva de ordem n de -1)
- **K =** 4
- **l =** 3
- **gamma =** 523776
- **alfa =** 261888
- **eta =** 6
- **beta =** 325

1.2 KeyGen

- Gerar A em Rq^{kxl}
- Gerar s1 que pertence a S_{eta} l e s2 que pertence a S_{eta} k , sendo S_{eta} o conjunto dos polinómios de modo a que a norma infinito do polinómio é menor que eta
- Seja t = A x s1 + s2
- A chave pública é (A,t) e a chave privada é (s1,s2)

1.3 Sign

- Gerar y que pertence a $S_{gamma-1}$ l
- w = HighBits(Ay,2alfa)
- c = H(m || w)
- z = y + c x s1

- Equanto a norma infinito de z for maior que gamma - beta e a norma infinito de LowBits(Ay - cs2,2alfa) for maior do que alfa - beta, voltamos ao início

[1]:
```
class Dilithium:
    def __init__(self):
        self.n=256
        self.q=2^23-2^13+1
        self.h=60
        self.r=1753
        self.k=4
        self.l=3
        self.gama=523776
        self.alfa=261888
        self.eta=6
        self.beta=325

    def keygen(self):
        Zx.<x>= ZZ[]
        Zq.<z>= PolynomialRing(GF(self.q))
        Rq.<z>= Zq.quotient(z^self.n+1)
        R.<x>= Zx.quotient(x^self.n+1)

        #Processo para gerar a matriz A

        K=[]
        for i in range(self.k*self.l):
            K.append(Rq.random_element())
        A= matrix(Rq,self.k,self.l,K)

        #Gerar s1 e s2

        S1=self.sam(self.eta,self.l) #S com tamanho lx1 em que as componentes␣
↪de s1 são polinómios
        #de Rq com norma menor que eta
        S2=self.sam(self.eta,self.k) #S com tamanho kx1

        t= A*S1+S2 #com tamanho 4x1

        pubKey=(A,t)
        privKey=(S1,S2)

        return pubKey,privKey

    def sign(self,pubKey,privKey,m):
        A=pubKey[0]
        s1=privKey[0]
        s2=privKey[1]
```

```
        Zq.<z>= PolynomialRing(GF(self.q))
        Rq.<z>= Zq.quotient(z^self.n+1)

        y=self.sam(self.gama-1,self.l)
        Ay=A*y

        w=self.HBpol(Ay)  #Cálculo dos HightBits de Ay

        # Cálculo do argumento para a função de hash

        string=''
        k=m[2:]
        string=string+k
        for i in range(len(w)):
            for j in range(len(w[i])):
                k=bin(w[i][j])
                if w[i][j]>=0:
                    string=string + k[2:]
                if w[i][j]<0:
                    string=string + k[3:]

        #cálculo do Hash

        c=self.Hash(string)
        cq=Rq(c)

        z=y+cq*s1

        while int(self.norma_inf_pol(z)[0])>=self.gama-self.beta and self.
↪norma_inf_pol(self.LBpol(Ay-cq*s2))>= self.alfa-self.beta:

            y=self.sam(self.gama-1,self.l)
            Ay=A*y
            w=self.HBpol(Ay)
            #string
            string=''
            k=m[2:]
            string=string+k
            for i in range(len(w)):
                for j in range(len(w[i])):
                    k=bin(w[i][j])
                    if w[i][j]>=0:
                        string=string + k[2:]
                    if w[i][j]<0:
                        string=string + k[3:]
            c=self.Hash(string)
            cq=Rq(c)
```

```
            z=y+cq*s1

        return (z,c)

    def verify(self,pubKey,m,sig):
        Zq.<z>= PolynomialRing(GF(self.q))
        Rq.<z>= Zq.quotient(z^self.n+1)

        A=pubKey[0]
        t=pubKey[1]

        z=sig[0]
        c=sig[1]
        cq=Rq(c)

        w=self.HBpol(A*z-cq*t)

        string=''
        k=m[2:]
        string=string+k
        for i in range(len(w)):
            for j in range(len(w[i])):
                k=bin(w[i][j])
                if w[i][j]>=0:
                    string=string + k[2:]
                if w[i][j]<0:
                    string=string + k[3:]

        #cálculo do Hash

        Hash=self.Hash(string)

        if self.norma_inf_pol(z)[0]>=self.gama-self.beta:
            print 'Assinatura rejeitada'
        else:
            print 'Assinatura válida'

    def Decompose(self,c,t):
        r=mod(c,self.q)
        r0=int(mod(r,int(t)))
        if r0>t/2:
            r0=r0-int(t)
        if r-r0==self.q-1:
            r1=0
            r0=r0-1
        else:
            r1=(r-r0)/(int(t))
```

```
        return (r1,r0)

    def HighBits(self,c):
        x=self.Decompose(c,2*self.alfa)
        return x[0]

    def LowBits(self,c):
        x=self.Decompose(c,2*self.alfa)
        return x[1]

    def HBpol(self,pol):
        k=pol.list()
        for i in range(len(k)):
            h=k[i]
            h=h.list()
            for j in range(len(h)):
                h[j]=self.HighBits(int(h[j]))
            k[i]=h
        return k

    def LBpol(self,pol):
        k=pol.list()
        for i in range(len(k)):
            k[i]=self.LowBits(k[i])
        return k

    def sam(self,lim,tam):
        Zq.<z>= PolynomialRing(GF(self.q))
        Rq.<z>= Zq.quotient(z^self.n+1)
        S=[]
        for i in range(tam):
            pol=[]
            for j in range(self.n):
                pol.append(randint(1,lim))

            S.append(Rq(pol))
        S=matrix(Rq,tam,1,S)
        return S

    def Hash(self,val):
        H=[]
        contador=0
        contador_num=0
        for i in range(0,self.n,2):
            u=val[i]+val[i+1]
            contador+=1
            if u=='11':
```

```
                H.append(0)
            if u=='01':
                H.append(1)
                contador_num+=1
            if u=='00':
                pass
            if u=='10':
                H.append(-1)
                contador_num+=1
            if contador_num>=self.h:
                break
        for i in range(self.n-contador):
            H.append(0)
        return H

    def norma_infinito(self,pol,n):

        #R=self.power2round(pol,n)
        J=pol.list()
        for i in range(len(J)):
            k=J[i]
            K=k.list()
            for j in range(len(K)):
                K[j]=abs(int(K[j]))
            J[i]=K
        L=[]
        for i in range(len(J)):
            L.append(max(J[i]))

        return max(L)

    def norma_inf_pol(self,vetor):
        Zq.<z>= PolynomialRing(GF(self.q))
        Rq.<z>= Zq.quotient(z^self.n+1)
        for i in range(vetor.nrows()):
            norm=self.norma_infinito(vetor[i],self.q)
            vetor[i]=norm
        return max(vetor)

    def power2round(self,w,n):
        Zx.<x> = ZZ[]
        r = (n-1)//2
        return Zx(map(lambda x: lift(x + r) - r , w.list()))
```

[2]:
```
D=Dilithium()
pub,priv=D.keygen()
x=D.sign(pub,priv,bin(24523))
```

```python
D.verify(pub,bin(24523),x)
```

```
Assinatura válida
```

[ ]:

```python

```