NewHope

May 23, 2020

0.1 NewHope

Para a implementação do esquema NewHope foi necessária uma análise cuidada do(s) documento(s) fornecidos pelo professor. A construção do que é pedido no enunciado, tanto do IND-CPA-KEM como do IND-CCA-PKE, tem por base implementar o **IND-CPA-PKE**. Para tal recorreu-se a funções auxiliares bem como às funções que se encontravam no documento:

- **hamming_weight**: Algoritmo do peso de Hamming
- **ntt**: Transformada
- **ntt_inv**: Transformada inversa
- **bitRev**: Reversão dos bits(bit-reversal)
- **PolyBitRev**: Reversão dos bits para polinómios
- **coefMult**: Multiplica dois polinómios em termos de coeficiente
- **Sample**: Amostra do segredo e erro do R-LWE
- **GenA**: Geração de uma distribuição polinomial aleatória em Rq
- **EncodePoly**: Codifica um polinomial em Rq num array de bytes
- **DecodePoly**: Descodifica um array de bytes num polinomial em Rq
- **EncodePK**: Codifica a chave pública
- **DecodePK**: Descodifica a chave pública
- **EncodeMsg**: Codifica a mensagem num polinomial em Rq
- **DecodeMsg**: Desodifica a mensagem num polinomial em Rq
- **Compress**: Comprime a mensagem a enviar
- **EncodeC**: Codifica o criptograma
- **DecodeC**: Descodifica o criptograma
- **Descompress**: Descomprime a mensagem para recuperar os dados
- **KeyGen**: Geração da chave pública e privada
- **encrypt**: Encripta a mensagem e devolve um criptograma
- **decrypt**: Desencripta o criptograma.

Para a passagem para o IND-CPA-KEM foram implementadas as seguintes funções: **keyGenKEM**, **encapsulationKEM** e **decapsulationKEM** que utilizam as funções **KeyGen**, **encrypt** e **decrypt**,respetivamente.

Para a passagem para o IND-CCA-PKE seria necessário recorrer à transformação de Fujisaki Okamoto, no entanto, não foi implementado.

[28]:
```
import hashlib
import random
import math

def hamming_weight(num):
        weight = 0

        while num:
            weight += 1
            num &= num - 1
        return weight

class NewHope(object):

    def __init__(self, n=128):
        if not any([n == t for t in [32,64,128,256,512,1024,2048]]):
            raise ValueError("improper argument ",n)
        self.n = n
        self.q = 1 + 2*n
        while True:
            if (self.q).is_prime():
                break
            self.q += 2*n

        print(self.q)
        self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen(); self.w = w

        g = (w^n + 1)
        xi = g.roots(multiplicities=False)[-1]
        self.xi = xi
        rs = [xi^(2*i+1)  for i in range(n)]
        self.base = crt_basis([(w - r) for r in rs])

    def sum_arrays(first, second):
        return [x + y % self.q for x, y in zip(first, second)]

    def sub_arrays(first, second):
        return [x - y % self.q for x, y in zip(first, second)]

    def ntt(self,f):
```

```
        def _expand_(f):
            u = f.list()
            return u + [0]*(self.n-len(u))

        def _ntt_(xi,N,f):
            if N==1:
                return f
            N_ = N/2 ; xi2 =  xi^2
            f0 = [f[2*i]   for i in range(N_)] ; f1 = [f[2*i+1] for i in␣
↪range(N_)]
            ff0 = _ntt_(xi2,N_,f0) ; ff1 = _ntt_(xi2,N_,f1)

            s  = xi ; ff = [self.F(0) for i in range(N)]
            for i in range(N_):
                a = ff0[i] ; b = s*ff1[i]
                ff[i] = a + b ; ff[i + N_] = a - b
                s = s * xi2
            return ff

        return _ntt_(self.xi,self.n,_expand_(f))

    def ntt_inv(self,ff):
        return sum([ff[i]*self.base[i] for i in range(self.n)])


    def GenA(self,seed):
        a = [None] * self.n
        extseed = bytearray(random.sample(range(0, 255), 33))
        extseed[0:32] = seed[0:32]
        for i in range(0, int(self.n/64)):
            ctr = 0
            extseed[32] = i
            state = hashlib.sha256(extseed).digest()
            while ctr < 64:
                buf = hashlib.sha256(state).digest()
                j = 0
                while j<168 and ctr<64:
                    val = buf[j] | (buf[j+1] << 8)
                    if val < 5*self.q:
                        a[i*64+ctr] = val
                        ctr += 1
                    j += 2
        return a

    def bitRev(self, v):
        s = 0
        for i in range(0, int(math.log(self.n,2))):
```

```
            s += (((v >> i) & 1) << (math.log(self.n,2)-1-i))
        return s

    def PolyBitRev(self, p):
        r  = [None] * self.n
        for i in range(0, self.n):
            r[self.bitRev(i)] = p[i]
        return r

    def Sample(self, seed, nounce):
        r = [None] * self.n
        extseed = bytearray(random.sample(range(0, 256), 34))
        extseed[0:32] = seed[0:32]
        extseed[32] = nounce
        for i in range (0,int(self.n/64)):
            extseed[33] = i
            buf = hashlib.shake_256(extseed).digest(int(128))
            for j in range(0,64):
                a = buf[2*j]
                b = buf[2*j+1]
                val = hamming_weight(a) + self.q - (hamming_weight(b) % self.q)
                r[i*64+j] = val
        return r

    def coefMult(self, c1, c2):
        r = [None] * self.n
        for i in range(0, self.n):
            r[i] =  (c1[i]*c2[i] % self.q)
        return r


    def EncodeC(self,u, h):

        c[0:((7*self.n)/4)-1] = EncodePoly(u)
        c[((7*self.n)/4): int((7*self.n)/4) + int((3*self.n)/8)] = h
        return c

    def EncodePoly(self,s):
        r = bytearray(random.sample(range(0, 256), int((7*self.n)/4)))

        for i in range(0, int((self.n/4))):
            t0 = s[(4*i)+0] % self.q
            t1 = s[(4*i)+1] % self.q
            t2 = s[(4*i)+2] % self.q
            t3 = s[(4*i)+3] % self.q
            r[(7*i)+0] = t0 & int(0xff)
            r[(7*i)+1] = (t0 >> 8) | (t1 << 6) & int(0xff)
```

```python
            r[(7*i)+2] = (t1 >> 2) & int(0xff)
            r[(7*i)+3] = (t1 >> 10) | (t2 << 4) & int(0xff)
            r[(7*i)+4] = (t2 >> 4) & int(0xff)
            r[(7*i)+5] = (t2 >> 12) | (t3 << 2) & int(0xff)
            r[(7*i)+6] = (t3 >> 6) & int(0xff)
        return r

    def EncodePK(self,b_hat, publicseed):
        r = [None]*(int((7*self.n)/4) + 32)
        r[0:int((7*self.n)/4)-1] = EncodePoly(b_hat)
        r[int((7*self.n)/4):int(((7*self.n)/4)+32)] = publicseed
        return r

    def EncodeMsg(self,m):
        v = [None]*self.n
        for i in range(0, 32):
            for j in range(0, 8):
                mask = -(((m[i]>>j))&1)
                v[(8*i)+j+0] = (mask&(int(self.q/2)))
                v[(8*i)+j+256] = (mask&(self.q/2))
                if self.n==1024:
                    v[(8*i)+j+512] = (mask&(int(self.q/2)))
                    v[(8*i)+j+768] = (mask&(int(self.q/2)))
        return v

    def DecodeC(self,c):
        u = DecodePoly(c[0:int((7*self.n)/4)-1])
        h = c[int((7*self.n)/4): int((7*self.n/4)+(3*self.n/8))]
        return u, h

    def DecodePoly(self,v):
        for i in range(0,int(self.n/4)):
            r = [None]*self.n
            r[(4*i)+0] = int(v[(7*i)+0]) | (((int(v[(7*i)+1])&int(0x3f))<<8))
            r[(4*i)+1] = (int(v[(7*i)+1]) >> 6) | ((int(v[(7*i)+2]) << 2)) |␣
↪(((int(v[(7*i)+3])&int(0x0f))))
            r[(4*i)+2] = (int(v[(7*i)+3]) >> 4) | ((int(v[(7*i)+4]) << 4)) |␣
↪(((int(v[(7*i)+5])&int(0x03))<<12))
            r[(4*i)+3] = (int(v[(7*i)+5]) >> 2) | ((int(v[(7*i)+6]) << 6))
        return r

    def DecodeMsg(self,v):
        m = bytearray(random.sample((range(0, 256)), 32))
        for i in range(0,256):
            t = abs((v[i+0]) % self.q - (int(self.q/2)))
            t = t + abs((((v[i+256]) % self.q) - (int(self.q)/2)))
            if self.n == 1024:
```

```
                t = t + abs((((v[i+512])% self.q) - (int(self.q)/2)))
                t = t + abs((((v[i+768])% self.q) - (int(self.q)/2)))
                t = t - self.q
            else:
                t = t - int(self.q/2)

            t = t >> 15
            m[i>>3] = m[i>>3] | (t<<(i&7))
        return m

    def DecodePK(self,pk):

        b_hat = DecodePoly(pk[0:int((7*self.n)/4)-1])
        seed = pk[int((7*self.n)/4): int(((7*self.n)/4)+32)]

        return b_hat, seed

    def Compress(self,v):
        k = 0
        t = bytearray(random.sample(range(0, 256), 8))
        h = bytearray(random.sample(range(0, 256), (int(3*self.n/8))))
        for l in range(0, int(self.n/8)):
            i = 8*l
            for j in range(0, 8):
                t[j] = v[i+j] % self.q
                t[j] = int((((t[j]<<3))+self.q/2)/self.q) & 7
            h[k+0] = (t[0] | ((t[1]<<3)) | ((t[2]<<6)))
            h[k+1] = ((t[2]>>2) | ((t[3]<<1)) | ((t[4]<<4)) | ((t[5]<<7)))
            h[k+2] = ((t[5]>>1) | ((t[6]<<2)) | ((t[7]<<5)))
            k += 3
        return h

    def Decompress(self,h):
        k = 0
        r = [None]*self.n
        for l in range(0, int(self.n/8)):
            i = 8*l
            r[i+0] = h[k+0] & 7
            r[i+1] = (h[k+0]>>3) & 7
            r[i+2] = (h[k+0]>>6) | (((h[1]<<2))&4)
            r[i+3] = (h[k+1]>>1) & 7
            r[i+4] = (h[k+1]>>4) & 7
            r[i+5] = (h[k+1]>>7) | (((h[2]<<1))&6)
            r[i+6] = (h[k+2]>>2) & 7
            r[i+7] = (h[k+2]>>5)
            k += 3
            for j in range(0, 8):
```

```
                r[i+j] = (((r[i+j])*self.q)+4)>>3
        return r

    def keyGen(self):
        seed = bytearray(random.sample(range(0, 256), 32))
        z = hashlib.sha512(b'0x01'+seed).digest()
        publicseed = z[0:32]
        noiseseed = z[32:64]
        _a = self.GenA(publicseed)
        s = self.PolyBitRev(self.Sample(noiseseed,0))
        e = self.PolyBitRev(self.Sample(noiseseed,1))
        _s = self.ntt(self.R(s))
        _e = self.ntt(self.R(e))
        aux =  self.coefMult(_a, _s)
        _b = sum_arrays(aux, _e)
        pk = EncodePK(_b,publicseed)
        sk = EncodePoly(_s)
        return pk, sk

    def encrypt(self, pk, message, coin):
        _b, publicseed = DecodePK(pk)
        _a = self.GenA(publicseed)
        _s = self.PolyBitRev(self.Sample(coin,0))
        _e = self.PolyBitRev(self.Sample(coin,1))
        __e = self.Sample(coin,2)
        _t = self.ntt(self.R(_s))
        _u = sum_arrays(self.coefMult(_a, _t), self.ntt(self.R(_e)))
        v = EncodeMsg(message)
        _v = sum_arrays(self.ntt_inv(self.coefMult(_b, _t)).list(), __e)
        print(_v)
        h = Compress(_v)
        c = encodeC(_u,h)
        return c

    def decrypt(self, criptogram, sk):
        _u, h = DecodeC(c)
        _s = DecodePoly(sk)
        _v = Descompress(h)
        x = self.ntt_inv(self.coefMult(_u, _s))
        print(x.list())
        res = sub_arrays(_v,x)

        m = Decode(res)
        return m

    #CPA-KEM
    def keyGenKEM(self):
```

```python
        pk, sk = self.keyGen()
        return pk, sk

    def encapsulationKEM(self,pk):
        coin = bytearray(random.sample(range(0, 256), 32))
        k , _coin = hashlib.shake_256(b'0x02'+coin).digest(int(64))
        c = self.encrypt(pk,k,_coin)
        ss = hashlib.shake_256()
        return c,ss

    def decapsulationKEM(c,sk):
        _k = self.decrypt(c,sk)
        ss = hashlib.shake_256(_k).digest(int(32))
        return ss
```

[ ]:
```python
a = NewHope()
m = a.R([1,2,3,4])
print(m)
pk, sk = a.keyGen()
```

[ ]:
```python
c = a.encrypt(pk, m, bytearray(random.sample(range(0, 256), 32)))
```

[ ]:
```python
m = a.decrypt(c,sk)
```

[ ]:
```python
pk, sk = keyGenKEM()
c, ss = encapsulationKEM(pk)
ss = decapsulationKEM(c,sk)
```