UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

ESTRUTURAS CRIPTOGRÁFICAS

Trabalho 3

Grupo 10

Autores:

Joel Gama (A82202)

Tiago Pinheiro (A82491)

22 de Maio de 2020

Conteúdo

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Estruturas Criptográficas integrada no perfil de Criptografia e Segurança da Informação.

Este trabalho é o terceiro de cinco trabalhos que ainda irão ser abordados nesta UC. O trabalho vai ter 6 implementações diferentes de dois esquemas algoritmos.

Primeiro era pedida a implementação do *KEM* tanto para o *NTRU-Prime* como para o *New Hope*.

O segundo exercício era para implementar o *KEM-IND-CPA* e o *PKE-IND-CCA* para ambos os esquemas.

NTRU-Prime

2.1 Implementação *KEM*

2.1.1 Parâmetros

Gerar o p,q e w

Nesta parte são gerados os principais parâmetros do código, o *p* e o *q* são ambos primos e maiores que 120, enquanto o *w* é inicializado como a divisão inteira de *p* por 4. Depois de inicializado, o *w* é passado a sua função de verificação para garantir que este cumpre os requisitos pedidos. A variável *d* é só uma variável auxiliar para os *round*.

```
p = next_prime(120)
q = next_prime(120)
w = p//4
d = (q-1)/2

w = verifyW(p,q,w,4)
```

Verificar o w

A função de verificar é bastante simples, caso o *w* não cumpra o primeiro requisito `2*p >= 3*w` é gerado um *w* mais pequeno. Se já cumprir o requisito, passámos à segunda condição `q >= 16*w +1` e, se ainda não cumprir, é gerado um *w* ainda mais pequeno. Caso o *w* cumpra esta condição sabemos que ele cumpre a outra.

```
def verifyW(p, q, w, indice):
    while (2*p < 3*w):
        indice = indice + 1
        w = p//indice
```

```
    while (q < (16*w + 1)):
        indice = indice + 1
        w = p//indice

    return w
```

Gerar *Polynomial Rings*

Nesta parte são gerados podes os *Polynomial Rings* e todos os *QuotientRing* que são precisos no código. Um ênfase para a verificação da irredutibilidade da equação `x^p-x-1`.

```
R_.<x>  = ZZ[]
R       = R_.quotient(x^p-x-1)

R3_.<x> = GF(3)[]
R3      = R3_.quotient(x^p-x-1)

Rq_.<x> = GF(q)[]
Rq      = Rq_.quotient(x^p-x-1)

if (x^p-x-1).is_irreducible() is False:
    print("Error\n")
```

2.1.2 Funções auxiliares

roundNext3

```
def round_next_3(inp,pol=None):
    try:
        inp = lift(inp).list()
    except:
        pass
    def f(u):
        a = lift(u) ; a = a if a <= d else a-q
        return 3*round(a/3)

    pr = [f(u) for u in inp]
    if pol:
        return pol(pr)
    else:
        return pr
```

round3

```
def round3(inp,pol=None):
    try:
        inp = lift(inp).list()
    except:
        pass
    def f(a):
        u = lift(a) ; u = u if u <= d else u-q
        u = u%3
        return u if u < 2 else -1

    pr = [f(a) for a in inp]
    if pol:
        return pol(pr)
    else:
        return pr
```

R3 to small

Para um polinómio de *R/3* para *small*. A estrutura auxiliar é porque no *lift* tinha o problema de não levantar elementos depois do último exponente. Ou seja, se o **p** fosse igual a 127 mas o valor do último exponente do polinómio fosse 120, o array *r* apenas iria ficar com 120 elementos.

```
def R3_to_small(inp):

    inp2 = lift(inp).list()
    anp = [0]*p

    for i in range(len(inp2)):
        anp[i] = inp2[i]

    def f(u):
        return u if u < 2 else -1
    return [f(u) for u in anp]
```

small

Cria um vetor de tamanho *p* escolhendo aleatoriamente elementos da lista `{1,-1,0}`.

```
def small(P=p):
    u = [rn.choice([-1,0,1]) for i in range(p)]
    return u
```

smallW

Cria um vetor de tamanho *p* escolhendo aleatoriamente **w** elementos da lista `{1,-1}`, depois adiciona `p-w` zeros e no final faz *shuffle*.

```
def smallW(P=p,W=w):
    u = [rn.choice([-1,1]) for i in range(w)] + [0]*(p-w)
    rn.shuffle(u)
    return u
```

VerifyG

Função que verifica se o vetor *small* **g** é invertível em **R/3**. É uma função recursiva que apenas para quando gerar, aleatoriamente, um vetor invertível.

```
def verifyG():
    while True:
        g = small()
        if(R3(g).is_unit()):
            break

    return g
```

pesosR

Função que avalia quantos "pesa"um vetor, ou seja, quantos elementos `1` ou `-1` tem um vetor.

```
def pesosR(vec):
    cont = 0

    for i in range(len(vec)):
        if (vec[i] != 0):
           cont = cont+1

    return cont
```

2.1.3 *Key generation*

Para gerar a chave pública, primeiro são gerados dois vetores, um *small* (**g**) e outro *small com pesos* (**f**). Neste caso, o **f** vai ter *w* elementos diferentes de zero, enquanto que o **g** vai ter um número aleatório de valores diferentes de zero.

De seguida, o vetor **f** é passado para *R/q*. Depois, calcula-se o vetor **g1** é que o inverso do vetor **g** em *R/3*.

Por fim, calcula-se a chave pública, onde se divide o vetor **g** em *R/q* por **3f**.

Como *output*, temos um dicionário que leva um polinómio **f** em *R/q*, o polinómio `1/g` em *R/3* e a chave pública, **h**.

```
def KeyGen():
    g = verifyG()
    f = smallW(p,w)
    F = Rq(f)

    g1 = 1/R3(g)
    h = Rq(g)/(3*F)

    return {'f': F, 'g1': g1, 'pk' : h}
```

2.1.4 *Encapsulation*

Para se fazer o *encapsulate* apenas é preciso um parâmetro, a chave pública.

Primeiro, calcula-se o **r**, que é um *small* com peso *w*. Depois, a partir do *r* e da chave pública calculámos o *c*, que vai fazer usado para confirmar o desencapsulamento.

Numa segunda parte, é feita a *hash* do vetor **r**, que é passado para string, disso resulta o **C**(confirmação) e o **K**(key).

O *output* será um dicionário com o **c**, a cofirmação e a *key*.

```
def Encapsulate(pk):
    r = smallW(p,w)
    c = round_next_3(pk*Rq(r))

    fhash = hashlib.sha512()
    fhash.update(str(r).encode('utf-8'))
    divisao = fhash.digest()

    C = divisao[:32]
    K = divisao[32:]

    return {'C': C, 'c': c, 'K': K}
```

2.1.5 *Decapsulation*

A função de desencapsulação recebe 4 parâmetros: a confirmação, o *c*, o vetor *f* e o polinómio *g*.

Na primeira parte, calcula-se o produto de `3fc` em *R/q* e o resultado é multiplicado pelo polinómio *g*, em *R/3*. Depois apenas se passa o polinómio *e* para array de `-1,1,0`.

Na segunda parte é feita a mesma coisa que na segunda parte do *Encapsulate*.

Na parte final da função são feitas as primeiras verificações, ou seja, verifica-se se o peso do vetor *r1* é igual a *w* caso não seja, não foi obtido sucesso. Depois, verifica-se se a confirmação é igual à confirmação passada pela função *Encapsulate*. Se tudo se verificar, passa-se o *KLinha* e o *r1* à função de teste para fazer as últimas verificações.

```
def Decapsulate(C,c,f,g1):
    a = round3(Rq(3*f)*Rq(c))
    e = R3(a)*g1
    r1 = R3_to_small(e)

    fhash = hashlib.sha512()
    fhash.update(str(r1).encode('utf-8'))
    divisao = fhash.digest()

    CLinha = divisao[:32]
    KLinha = divisao[32:]

    if (pesosR(r1) == w):
        if(CLinha == C):
            return {'r1': r1, 'k': KLinha}
        else:
            return False
    else:
        return False
```

2.1.6 Test

Uma função muito simples que gera a chaves, faz o encapsulamento com a *public key* e depois chama a função de desencapsulamento para retornar o *segredo*. No final, verifica se o *c* é igual, ou seja, a multiplicação da chave pública pelo *r*, e ainda, se a *key* for igual retorna *True*.

```
def run():
    keys   = KeyGen()
    crypto = Encapsulate(keys['pk'])
    decryp = Decapsulate(crypto['C'],crypto['c']
                                    ,keys['f'],keys['g1'])
    if (crypto['c'] ==
                round_next_3(keys['pk']*Rq(decryp['r1']))):
        return base64.b64encode(crypto['K'])==
                            base64.b64encode(decryp['k'])
    else:
        return false
```

NewHope

3.1 Implementação Base

Como forma de implementação do *NewHope* começamos por implementar a versão *NewHope_CPA_PKE*.

3.1.1 Parâmetros

Começamos pelos parâmetros utilizados. Definimos um n, q e k (parâmetros da distribuição do ruído) e por fim *T* um objeto da classe *NTT*. Depois é necessário definir os anéis a utilizar e criar geradores aleatórios.

```
n = 1024
q = 12289
k = 8  #noise

T = NTT(n)


## Os anéis usuais com o módulo ciclotómico
R_.<w> = ZZ[]
R.<x>  = QuotientRing(R_,R_.ideal(w^n+1))

Rq_.<w> = GF(q)[]
Rq.<x>  = QuotientRing(Rq_,Rq_.ideal(w^n+1))


# Geradores aleatorios
def random_pol(n=n):
    p = R_.random_element(n)
    return Rq(p)

def chi(k=k):
    samples = range(2*k)
    return sum([random.choice([-1,1]) for _ in samples])
```

```
def random_err(n=n):
    return Rq([chi() for _ in range(n)])

def random_seed(n):
    seed = [randint(0,1) for i in range(n)]
    seed = a2b(seed)

    return seed
```

3.1.2 Geração de chaves

O método utilizado para criar as chaves pública e privada a utilizar começa por gerar uma *seed* aleatória com tamanho 32 *bytes*, a partir da qual em seguida, utilizando a função de *hash* criptográfica *Shake256*, se gera um novo valor com o dobro do tamanho. A partir do resultado do *shake256* são criadas duas novas *seeds* uma para a chave pública e outra para ser utilizada para gerar os polinómios a utilizar.

Depois de ter as *seeds* é gerado um polinómio com a *publicseed* e dois *samples* a partir da *noiseseed*. Cada polinómio, exceto o *a*, é convertido para *NTT*. Por fim é feito o calculo de *b*, que em conjunto com a *publicseed* formam a chave pública.

O método retorna um par de chave pública e chave privada (o primeiro polinómio gerado a partir da *noiseseed*.

```
# Gerar chaves
def keyGen():

    shake = hashlib.shake_256()
    seed = random_seed(32)
    x = 64
    n = int(x)

    shake.update(seed)
    z = shake.digest(n)

    publicseed = seed  #z[:32]
    noiseseed  = seed1 #z[32:]

    a  = genA(publicseed)
    s  = polyBitRev(sample(noiseseed,0))
    sk = T.ntt(s)
    e  = polyBitRev(sample(noiseseed,1))
    ee = T.ntt(e)

    b = a * sk + ee

    pk = (b,publicseed)

    return (pk,sk)
```

3.1.3 *Encrypt* e *Decrypt*

No método $encrypt$ é onde o conteúdo é cifrado. A partir de 3 polinómios em *NTT* são calculados os dados necessários para o resultado.

Depois de ter os polinómios é feito o *encode* da mensagem. Em seguida é calculado o $v = T.ntt\_inv(b * t) + ee + v$, que depois de comprimido é utilizado para calcular o *encode* do *ciphertext*.

No $decrypt$ é feito o processo contrário até a obtenção da mensagem.

```
def encrypt(pk,m,seed):

    (b,publicseed) = pk

    a  = genA(publicseed)
    s  = polyBitRev(sample(seed,0))
    e  = polyBitRev(sample(seed,1))
    ee = sample(seed,2)

    t  = T.ntt(s)
    u  = a * t + T.ntt(e)
    v  = encode(m)
    vv = T.ntt_inv(b * t) + ee + v
    h  = compress(vv)
    c  = encodeC(u,h)

    return c

def decrypt(c,sk):

    (u,h) = decodeC(c)
    v = decompress(h)
    m = decode(v -  T.ntt_inv(u * sk))

    return m
```

3.1.4 Métodos auxiliares

GenA

O parâmetro público *a* é gerado a partir do método $genA$. É passado como input uma *seed* que é utilizada para os primeiros 32 *bytes* de uma nova *seed* utilizada no *shake128()*. O *shake128* é utilizado para gerar um *state* com 200 *bytes* e um *buf* com 168 *bytes* e 1 bloco. A partir desses parâmetros é calculado o *a*.

```
# Gerar polinómio a para gerar as chaves
def genA(seed):

    a = lift(random_pol())
    seed = b2a(seed)
    shake = hashlib.shake_128()
    extseed = seed[:31]
    x = 200
    n1 = int(x)
    y = 168
    n2 = int(y)

    for i in range((n/64) - 1):
        ctr = 0
        extseed.append(i)
        extseed = a2b(extseed)
        shake.update(extseed)
        state = shake.digest(n1)
        while ctr < 64:
            buf = shake.digest(n2)
            shake.update(state)
            state = shake.digest(n1)
            j = 0
            while (j < 168 and ctr < 64):
                val = buf[j]|(buf[j+1] << 8)
                if val < 5.q:
                    a[i*64+ctr] = val
                    ctr = ctr + 1
               j = j+2

    return a
```

Sample

O método $sample$ é utilizado para criar um polinómio em *Rq* a partir de uma *seed* e de um *noise*.

```
# Sample de polinómios
def sample(seed, nonce):

    shake = hashlib.shake_256()
    r = lift(random_pol())
    x = 128
    n = int(x)

    extseed = [64]
    extseed[:31] = seed
    extseed[32:] = nonce

    for i in range((n/64)-1):
        extseed.append(i)
        shake.update(extseed)
        buf = shake.digest(n)
        for j in range(63):
            a = buf[2 * j]
            b = buf[2 * j + 1]
            r[64*i+j]= HW(a) + q - Mod(HW(b),q)

    return r
```

Encode e Decode

Os métodos de $encode$ e $decode$ são utilizados para codificar e descodificar a mensagem a enviar. A mensagem é representada como um *array* de *bytes* e é codificada para um elemento em *Rq*. No processo de descodificação é feito o contrário.

Para garantir robustez contra erros a mensagem é codificada em *n/256* coeficientes.

```
# Encode da mensagem
def encode(m):
    v = lift(random_pol())
    for i in range(31):
        for j in range(7):
            mask = -((m[i]>>j)&1)
            v[8*i+j+0]   = mask&(q/2)
            v[8*i+j+256] = mask&(q/2)
            if n == 1024:
                v[8*i+j+512] = mask&(q/2)
                v[8*i+j+68]  = mask&(q/2)
    return v
```

```
# Decode da mensagem
def decode(v):

    v = lift(v)
    m = []
    for i in range(255):
        t = |Mod(v[i+0],q) - (q-1)/2|
        t = t + |Mod(v[i+256],q) - (q-1)/2|
        if n == 1024:
            t = t + |Mod(v[i+512],q) - (q-1)/2|
            t = t + |Mod(v[i+768],q) - (q-1)/2|
            t = t-q
        else:
            t = t - q/2
        t = t >> 15
        m[i>>3] = m[i>>3] or (t<<(i&7))

    return m
```

Encode e Decode Ciphertext

O método $encodeC$ faz o *encode* do *ciphertext* a partir da compressão de *v* (utilizando o método $compress$). O precesso contrário é realizado pelos métodos $decodeC$ e $decompress$.

```
# Encode do criptograma
def encodeC(u,h):

    u = lift(u)
    c[0:(7*n/4-1)] = u
    c[(7*n/4-1):(7*n/4+3*n/8-1)] = h

    return c

# Decode do criptograma
def decodeC(c):

    u = lift(random_pol())
    u = c[0:(7*n/4-1)]
    h = c[(7*n/4-1):(7*n/4+3*n/8-1)]

    return (u,h)
```

```
# Comprimir polinómio
def compress(v):

    k = 0
    t = []
    h = [3*n/8]
    v = lift(v)

    for l in range(n/8-1):
        i = 8*l
        for j in range(7):
            t[j] = Mod(v[i+j],q)
            t[j] = ((b2i(t[j] << 3) + q/2)/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 = k+3

    return h

# Descomprimir polinómio
def decompress(a):

    k = 0
    r = lift(random_pol())

    for l in range(n/8-1):
        i = 8*l

        r[i+0] = a[k+0]&7
        r[i+1] = (a[k+0] >> 3)&7
        r[i+2] = (a[k+0] >> 6)|((a[k+1] << 2)&4)
        r[i+3] = (a[k+1] >> 1)&7
        r[i+4] = (a[k+1] >> 4)&7
        r[i+5] = (a[k+1] >> 7)|((a[k+2] << 1)&6)
        r[i+6] = (a[k+2] >> 2)&7
        r[i+7] = (a[k+2] >> 5)

        k = k+3
        for j in range(7):
            r[i+j] = (r[i+j]*q+4)>>3

    return r
```

Métodos de conversão e cálculo

Foram desenvolvidos três métodos extra para fazer a conversão de *arrays* de inteiros para uma *string* binária e também para o processo contrário. Por fim, foi definido um método para o cálculo do *Hamming weight* utilizado no método $Sample$.

```
# Converter um array para binário
def a2b(array):

    binary = bytearray(array)

    return binary

# Converter de binário apra array
def b2a(binary):

    array = [d for d in binary[2:]]

    return array

# Hamming weight
def HW(x):
    s = 0

    for i in range(x):
        s = s + x[i]

    return s
```

3.2 Versão *KEM-IND-CPA*

Para a implementação do *KEM_IND_CPA* foram implementados os métodos para gerar as chaves de forma a não ter acesso ao algoritmo. Em seguida no encapsulamento é feito o *encrypt* e returnado o resultado. Por fim, no desencapsulamento, é feito o *decrypt*. Os métodos utilizados (*keyGen, encrypt e decrypt*) fazem parte da classe *NewHope_CPA_PKE* enquanto estes são definidos na classe *NewHope_CPA_KEM*.

```
def gen():
    (pk,sk) = keyGen()

    return (pk,sk)

def encapsulation(pk):

    seed = random_seed(32)
    shake = hashlib.shake_256()
    x = 32
    n = int(x)

    shake.update(seed)
    k = shake.digest(n)

    coin = shake.digest(n)

    c = encrypt(pk,k,coin)

    shake.update(k)
    ss = shake.digest(n)

    return (c,ss)

def decapsulation(c,sk):

    shake = hashlib.shake_256()
    x = 32
    n = int(x)

    k = decrypt(c,sk)

    shake.update(k)
    ss = shake.digest(n)

    return ss
```

3.3 Versão *PKE-IND-CCA*

Na implementação da versão *NewHope_CCA_PKE* é dito no artigo *NewHope* que se pode utilizar técnicas *standard* de conversão, especificadas pelo *NIST*, para converter a implementação de *NewHope_CCA_KEM* em *NewHope_CCA_PKE*.

Conclusão

Este trabalho foi, sem dúvida, o mais difícil realizado até ao momento. O grupo sentiu imensas dificuldades com os tipos e cálculos necessários a efetuar.

Foi possível aplicar conceitos relacionados *Criptosistemas pós-quânticos*, *PKE* e *KEM*. Mais especificamente os esquemas *NTRU-Prime* e *New Hope*.

Em ambos os casos, *NTRU-Prime* e *New Hope*, apenas foi possível concluir com sucesso o exercício 1, ou seja, a implementação do *KEM*.

No *NTRU-Prime* ficou por fazer o exercício 2, algo que o grupo quer concluir ainda no decorre dos próximos dias, se for possível.

Na parte do *New Hope*, foi feito o exercício 2 mas como tem erros, não foi possível atingir a solução final.