Trabalho Prático 2

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

NTRU

> NTRU is an open-source public-key cryptosystem that uses lattice-based cryptography to encrypt and decrypt data. It consists of two algorithms: NTRUEncrypt, which is used for encryption, and NTRUSign, which is used for digital signatures. Unlike other popular public-key cryptosystems, it is resistant to attacks using Shor's algorithm. NTRUEncrypt was patented, but it was placed in the public domain in 2017. NTRUSign is patented, but it can be used by software under the GPL.
>
> https://en.wikipedia.org/wiki/NTRU

Deste modo, iremos apresentar duas implementações (com recurso ao *SageMath* deste algoritmo: NTRU-PKE (que seja IND-CCA seguro) e NTRU-KEM (que seja IND-CPA seguro)

NOTA: as nossas implementações irão utilizar o seguinte documento como referência: https://ntru.org/f/ntru-20190330.pdf. Este documento já apresenta um PKE-IND-CCA e KEM-IND-CCA, pelo que desta forma só foi necessário implementar as duas versões especificadas nessa submissão.

NTRU-PKE

Para esta implementação (PKE-IND-CCA), teremos de seguir os seguintes passos:

1. Inicialização da classe, através de parâmetros de acordo com o NTRU-HPS. Além disso, são também criados os anéis necessários (Zx, R e Rq), sendo que os outros anéis como o Sq e o S3 não foram necessários, uma vez que se usou sempre os valores de forma arredondada (por exemplo, no caso do 3, em vez de serem valores entre {0,1,2}, usa-se valores entre {-1,0,1});
2. Geração da chave pública e da chave privada;
3. Métodos de cifragem e decifragem.

Geração das chaves

Assim, para a geração das chaves foi implementada uma função (`generate_keys(self,seed)`) que utiliza uma seed como parâmetro de entrada. De uma forma breve, tentamos explicar tudo o que foi necessário implementar:

- Seed: de bits aleatórios que serve para gerar os polinómios ternários `f` e `g`. Esta seed deve ter tamanho suficiente para posteriormente ser dividida a metade para gerar os polinómios `f` e `g`;
- Gerar `f` e `g`: (f,g) <- Sample_fg(seed). Esta função auxiliar pega na seed e gera um polinómio ternário `f` e `g`, gerados de forma simples:
  - Se o bit é igual a 0: dá-se o valor de 1 como coeficiente;
  - Se o bit é igual a 1: dá-se o valor de -1 como coeficiente.
  - Depois, e para que o polinómio tenha o tamanho certo, completa-se tudo com 0´s e faz-se o shuffle dos elementos da lista resultante;
- Uma vez gerados os polinómios `f` e `g`, pode-se então passar para a fase de cálculo dos elementos da chave privada (f,fp,hq) e da chave pública (h). Para tal são necessários alguns cálculos de inversas. **Neste cálculo, é necessário ter atenção que alguns podem não possuir inversa**. Caso tal aconteça, é necessário voltar a iterar e a gerar novos polinómios `f` e `g`. Resumidamente, os cálculos necessários são os seguintes:
  - fp <- (1/f) mod(3, phi(n))
  - fq <- (1/f) mod(q, phi(n))
  - h <- (3.g.fq) mod(q,phi(1)phi(n))
  - hq <- (1/f) mod(q, phi(n))

Cifragem

Foi implementada uma função (`cipher_text(self, h, rm)`) que utiliza a chave pública e o tuplo (r,m):

- A função cifra recebe como parâmetros a chave pública `h`, um `r` e um `m` que será a mensagem a enviar. O `r` é um parâmetro gerado de forma aleatória.
- De seguida, é calculado o criptograma através da expressão: c <- (r.h + m) mod(q, phi(n)). **Nota**: Normalmente, no lugar do `m`, deveria de estar o `Lift(m)`, mas de acordo com o NTRU-HPS, temos que `Lift(m) = m`;

Decifragem

Foi implementada uma função (`decipher_text(self, sk, c)`) que utiliza a chave privada (f,fp,hq) e ainda o criptograma:

- A função decifra recebe como parâmetros a chave privada (f,fq,hq) e o criptograma (c);
- a <- (c.f) mod(q,phi(1)phi(n));
- m <- (a.fp) mod(3,phi(n));
- r <- ((c-m).hq) mod(q,phi(n)), onde m é igual a Lift(m) = m, de acordo com o NTRU-HPS;
- Se os polinómios (r,m) não forem ternários, retorna (0,0,1), senão retorna (r,m,0).

In [ ]:
```
import random, hashlib
import numpy as np

# Baseado no esquema da página 25 do documento https://ntru.org/f/ntru-20190330.pdf
# https://latticehacks.cr.yp.to/ntru.html

class NTRU_PKE(object):

    def __init__(self, N=821, Q=4096, D=495, timeout=None):

        # Todas as inicializações de parâmetros são baseadas na submissão com os parâmetros do ntruhps4096821, onde n = 821 (página 29 do documento)
        self.n = N
        self.q = Q
        self.d = D

        # Definição dos anéis
        Zx.<x>  = ZZ[]
```

```
        self.Zx = Zx
        Qq = PolynomialRing(GF(self.q), 'x')
        x = Zx.gen()
        y = Qq.gen()
        R = Zx.quotient(x^self.n-1)
        self.R = R
        Rq = QuotientRing(Qq, y^self.n-1)
        self.Rq = Rq


    # Gera uma string de bits com tamanho size e d 1's
    def randomBitString(self, size):

        # Gera uma sequência de n bits aleatórios
        u = [random.choice([0,1]) for i in range(size)]
        # Mistura os valores da lista, só para aumentar a aleatoriedade
        random.shuffle(u)
        return u


    # Verifica se um polinómio é ternário
    def isTernary(self, f):
        res = True
        v = list(f)
        for i in v:
            if i > 1 or i < -1:
                res = False
                break
        return res


    # Produz o polinómio f mod q. Mas, em vez de ser entre 0 e q-1, fica entre -q/2 e q/2-1
    def round_mod(self, f, q):

        g = list(((f[i] + q//2) % q) - q//2 for i in range(self.n))
        return self.Zx(g)


    # Produz a inversa de um polinómio f mod x^n-1 mod p, em que p é um número primo.
    def inverse_modP(self, f, p):

        T = self.Zx.change_ring(Integers(p)).quotient(x^self.n-1)
        return self.Zx(lift(1 / T(f)))


    # Como a função de cima, mas o q aqui é uma potência de 2
    def inverse_mod2(self, f, q):

        assert q.is_power_of(2)
        g = self.inverse_modP(f, 2)
        while True:
            r = self.round_mod(self.R(g*f), q)
            if r == 1:
                return g
            g = self.round_mod(self.R(g*(2 - r)), q)


    # Gera um polinómio ternário
    def Ternary(self, bit_string):
```

```
        # cria um array
        result = []
        # Itera d vezes
        for j in range(self.d):
            # Se o bit for 0, acrescenta 1, senão -1
            if bit_string[j] == 0:
                result += [1]
            elif bit_string[j] == 1:
                result += [-1]
        # Preenche com 0's o array restante
        result += [0]*(self.n-self.d)
        # Mistura os valores do array
        random.shuffle(result)

        return self.Zx(result)


    # Gera um polinómio f em Lf (neste caso, em T+) e um polinómio g em Lg (neste caso, em {ϕ1.v : v € T+})
    def Sample_fg(self, seed):

        x = self.R.gen()

        # Parse de fg_bits em f_bits||g_bits
        f_bits = seed[:self.d]
        g_bits = seed[self.d:]

        # Definir f = Ternary_Plus(f_bits)
        f = self.Ternary(f_bits)
        # Definir g0 = Ternary_Plus(g_bits)
        g = self.Ternary(g_bits)

        return (f,g)


    # Gera um polinómio r em Lr (neste caso, em T) e um polinómio m em Lm (neste caso, em T)
    def Sample_rm(self, coins):

        # sample_iid_bits = 8*n - 8
        sample_iid_bits = 8*self.n - 8

        # Parse de rm_bits em r_bits||m_bits
        r_bits = coins[:sample_iid_bits]
        m_bits = coins[sample_iid_bits:]

        # Set r = Ternary(r_bits)
        r = self.Ternary(r_bits)
        # Set m = Ternary(m_bits)
        m = self.Ternary(m_bits)

        return (r,m)


    # Função usada para gerar o par de chaves pública e privada
    def generate_keys(self, seed):

        while True:
            try:
                (f,g) = self.Sample_fg(seed)
```

```
                # fp <- (1/f) mod(3;ϕn)
                fp = self.inverse_modP(f, 3)
                # fq <- (1/f) mod (q;ϕn)
                fq = self.inverse_mod2(f, self.q)
                # gq <- (1/f) mod (q;ϕn) (só para garantir que h e invertível)
                gq = self.inverse_mod2(g, self.q)
                # h <- (3.g.fq) mod(q;ϕ1ϕn)
                h = self.round_mod(3*self.R(g*fq), self.q)
                # hq <- (1/h) mod (q;ϕn)
                hq = self.inverse_mod2(h, self.q)
                break
            except:
                # Para que a nova iteração tenha uma nova seed
                seed = self.randomBitString(2*self.d)
                pass

        return {'sk' : (f,fp,hq) , 'pk' : h}


    # Recebe como parâmetros a chave pública e o tuplo (r,m)
    def cipher_text(self, h, rm):

        r = rm[0]
        m = rm[1]
        # c <- (r.h + m') mod (q,ϕ1ϕn)
        c = self.round_mod(self.R(h*r) + m, self.q)

        return c


    # Recebe como parâmetros a chave privada (f,fp,hq) e ainda o criptograma
    def decipher_text(self, sk, c):

        # a <- (c.f) mod(q,ϕ1ϕn)
        a = self.round_mod(self.R(c*sk[0]), self.q)
        # m <- (a.fp) mod(3,ϕn)
        m = self.round_mod(self.R(a * sk[1]), 3)
        # r <- ((c-m').hq) mod(q,ϕn)
        aux = (c-m) * sk[2]
        r = self.round_mod(self.R(aux), self.q)
        # Se os polinómios não forem ternários, retorna erro
        if not self.isTernary(r) and not self.isTernary(m):
            (0,0,1)
        return (r,m,0)
```

In [ ]:

```
# Parâmetros do NTRU (ntruhps4096821)
N=821
Q=4096
D=495

# Inicialização da classe
ntru = NTRU_PKE(N,Q,D)

print("[Teste da cifragem e decifragem]")
keys = ntru.generate_keys(ntru.randomBitString(2*D))
rm = ntru.Sample_rm(ntru.randomBitString(11200))

c = ntru.cipher_text(keys['pk'], rm)
```

```
rmDec = ntru.decipher_text(keys['sk'], c)

if rmDec[0] == rm[0] and rmDec[1] == rm[1] and rmDec[2] == 0:
    print("As mensagens e os r's são iguais!!!!")
else:
    print("A decifragem falhou!!!!")
```

```
[Teste da cifragem e decifragem]
As mensagens e os r's são iguais!!!!
```

NTRU-KEM

A inicialização da classe KEM aproveita a classe NTRU_PKE definida anteriormente para recorrer principalmente às funções `generate_keys`, `cipher_text` e `decipher_text` lá definidas. Também, são ainda definidos os parâmetros de segurança de acordo com o NTRU-HPS que também passados à classe PKE.

Assim, para esta implementação, foram seguidos os seguintes passos.

1. **Geração das chaves:**
   - Para a geração do par de chaves pública e privada, usou-se a função de geração de chaves definida na classe NTRU_PKE, sendo que deste modo já se consegue ter os parâmetros (f,fq,hq) e ainda o h;
   - Assim, basta gerar o parâmetro \`s\`, que é feito recorrendo a s <- {0,1}^256, ou seja, à geração de uma sequência de bits aleatórios.
2. **Encapsulamento e geração da chave:**
   - Criação de uma função que encapsula, recebendo como parâmetro a chave pública e produz o par (c,k), onde c é o 'encapsulamento' da chave e o k é a chave em si;
   - Depois é gerada uma sequência aleatória de bits: coins <-> {0,1}^256;
   - É gerado um polinómio ternário \`r\` e \`m\` de forma aleatória recorrendo à função: (r,m) <- Sample_rm(coins);
   - De seguida, procede-se à cifragem do (r,m): c <- Encrypt(h,(r,m)), sendo este c o 'encapsulamento' da chave;
   - Por fim, faz-se o hash de r e m para obter a chave simétrica: k <- H1(r,m).
3. **Desencapsulamento da chave:**
   - Criação de uma função que desencapsula, recebendo como parâmetros a chave secreta/privada e o encapsulamento da chave e retorna a chave simétrica k;
   - Primeiramente, é feito logo a decifragem do encapsulamento da chave c através da função definida no PKE anterior: (r,m,fail) <- Decrypt((f,fp,hq),c), lembrando que a chave secreta que esta função recebe não incluí o parâmetro \`s\`;
   - Faz-se o hash de r e m para obter a chave simétrica: k1 <- H1(r,m);
   - Faz-se o hash de s e c para obter uma outra chave diferente para o caso de o desencapsulamento falhar: k2 <- H2(s,c);
   - Se fail = 0, então retorna k1, senão retorna k2.

In [ ]:
```
# Baseado no esquema da página 25 do documento https://ntru.org/f/ntru-20190330.pdf
# https://latticehacks.cr.yp.to/ntru.html

class NTRU_KEM(object):

    def __init__(self, N=821, Q=4096, D=495, timeout=None):

        # Todas as inicializações de parâmetros são baseadas na submissao com os parâmetros do ntruhps4096821, onde n = 821
        self.n = N
        self.q = Q
        self.d = D
```

```
        # inicialização da instância NTRU_PKE
        self.pke = NTRU_PKE(self.n, self.q, self.d)


    #função para calcular o hash (recebe dois polinómios)
    def Hash1(self, e0, e1):

        ee0 = reduce(lambda x,y: x + y.binary(), e0.list() , "")
        ee1 = reduce(lambda x,y: x + y.binary(), e1.list() , "")
        m = hashlib.sha3_256()
        m.update(ee0.encode())
        m.update(ee1.encode())
        return m.hexdigest()


    #função para calcular o hash (recebe uma string de bits e um polinómio)
    def Hash2(self, e0, e1):

        ee1 = reduce(lambda x,y: x + y.binary(), e1.list() , "")
        m = hashlib.sha3_256()
        m.update(e0.encode())
        m.update(ee1.encode())
        return m.hexdigest()


    # Funão usada para gerar o par de chaves pública e privada(acrescenta ao geraChaves1() um s)
    def generate_keys(self, seed):

        # ((f,fp),h) <- KeyGen'()
        keys = self.pke.generate_keys(seed)
        # s <-$ {0,1}^256
        s = ''.join([str(i) for i in self.pke.randomBitString(256)])

        # return ((f,fp,hq,s),h)
        return {'sk' : (keys['sk'][0],keys['sk'][1],keys['sk'][2],s) , 'pk' : keys['pk']}


    # Funçã que serve para encapsular a chave que for acordada a partir de uma chave pública
    def encaps(self, h):

        # coins <-$ {0,1}^256
        coins = self.pke.randomBitString(256)
        # (r,m) <- Sample_rm(coins)
        (r,m) = self.pke.Sample_rm(self.pke.randomBitString(11200))
        # c <- Encrypt(h,(r,m))
        c = self.pke.cipher_text(h, (r,m))
        # k <- H1(r,m)
        k = self.Hash1(r,m)

        return (c,k)

    # Funçã usada para desencapsular uma chave, a partir do seu "encapsulamento" e da chave privada
    def desencaps(self, sk, c):

        # (r,m,fail) <- Decrypt((f,fp,hq),c)
        (r,m,fail) = self.pke.decipher_text((sk[0], sk[1], sk[2]), c)
        # k1 <- H1(r,m)
        k1 = self.Hash1(r,m)
```

```
        # k2 <- H2(s, c)
        k2 = self.Hash2(sk[3],c)
        # if fail = 0 return k1 else return k2
        if fail == 0:
            return k1
        else:
            return k2
```

In [ ]:

```
# Parametros do NTRU (ntruhps4096821)
N=821
Q=4096
D=495

# Inicialização da classe
ntru = NTRU_KEM(N,Q,D)

print("[Teste do encapsulamento e desencapsulamento]")
keys1 = ntru.generate_keys(ntru.pke.randomBitString(2*D))

(c,k) = ntru.encaps(keys1['pk'])
print("Chave = " + k)

k1 = ntru.desencaps(keys1['sk'], c)
print("Chave = " + k1)

if k == k1:
    print("A chave desencapsulada é igual à resultante do encapsulamento!!!")
else:
    print("O desencapsulamento falhou!!!")
```

```
[Teste do encapsulamento e desencapsulamento]
Chave = 09438f477b09369254c3a39460585cc160cb541e19f4215f773eae257b31384a
Chave = 09438f477b09369254c3a39460585cc160cb541e19f4215f773eae257b31384a
A chave desencapsulada é igual à resultante do encapsulamento!!!
```

7 Advantages and limitations

Our submission has a number of advantages.

- It is **correct**. The IND-CCA2 KEM always establishes a key; it never aborts because of a decryption failure. This simplifies the analysis of the scheme, and makes it an attractive drop-in replacement for KEMs that are in use today.
- It is **well studied**. Among the assumptions underlying post-quantum cryptosystems, the OW-CPA security of NTRU is well studied. NTRU, and similar systems, have frequently been used to benchmark new techniques in lattice reduction [37, 7, 15, 8]. This history of concrete cryptanalysis should inspire some confidence in NTRU. The tight reduction from the IND-CCA2 security of our KEM to the OW-CPA security of the ANTS'98 DPKE means that this history is relevant to the concrete security of our KEM.
- It is **flexible**. The underlying DPKE can be parameterized for a variety of use cases with different size, security, and efficiency requirements. We have discussed this in Section 2.4 and depicted some of the trade-offs in Figures 11, 12, and 13.
- It is **simple**. The DPKE has only two parameters, n and q, and can be described entirely in terms of simple integer polynomial arithmetic. The transformation to an IND-CCA2 secure KEM is conceptually simple.
- It is **fast**. ntruhrss701 was among the fastest submissions in the first round. We expect that this will remain true in the second round.
- It is **compact**. Our ntruhps2048677 parameter set achieves level one security with a wide security margin, level three security under a reasonable assumption, and has public keys and ciphertexts of only 930 bytes.
- It is **patent free**. The relevant patents have expired.

It also has several limitations.

- NTRU is unlikely to be the fastest submission, unlikely to be the most compact submission, and unlikely to be the most secure submission. However, it will be competitive on products of these measures.
- The choice of optimal parameters for NTRU is currently limited by a poor understanding of the non-asymptotic behavior of new algorithms for SVP. This is a limitation that is shared with all lattice based cryptosystems.
- There is structure in NTRU that is not *strictly* necessary, and this may also be seen as a limitation. It is possible to eliminate the structure of a sparse ternary secret at a cost in terms of correctness or compactness. It is also possible to eliminate the cyclotomic structure of the ring; comparisons with NTRU Prime will reveal the cost of doing so.