Universidade do Minho
Escola de Engenharia

Estruturas Criptográficas

Criptosistemas pós-quânticos PKE/KEM

Implementação esquemas KEM **NTRU-Prime** e **NewHope** em Python/SageMath

Submitted To:
José Valença
Professor Catedrático
Tecnologias da Informação e Segurança

Submitted By :
Diogo Araújo, A78485
Diogo Nogueira, A78957
Group 4

Conteúdo

1 Implementação do NTRU-Prime com o *SageMath*

1.1 Descrição do Exercício

A ideia do exercício passa por construir toda uma classe Python/SageMath que implemente o esquema KEM **NTRU-Prime**, algoritmo candidato ao **NIST's *Post-Quantum Cryptography Standardization Project***.

Com esse fim em mente, o grupo recorreu à documentação de candidatura do algoritmo fornecida pelo documento, mais especificamente ao documento de envio principal de seu nome **"NTRU Prime"**, submitido no dia 30/11/2017.

1.2 Descrição da Implementação

Com o *Primary Submission Document* em mão e com uma análise atenta de todo o processo algorítmico, podem-se definir o conjunto de definições que a classe Python terá e que permitirão no final fazer um pequeno teste em termos de resultados, para as versões PKE-IND-CCA e KEM-IND-CPA. De forma a faciliar a compreensão/execução de todas as etapas pensadas para o algoritmo, criou-se a divisão que consta no próprio documento em análise.

Com a pesquisa necessária e com a ideia do funcionamento do algoritmo em mente, estabelecem-se as seguintes implementações:

1. ***Parameter Space* - Criação/Geração dos Parâmetros e de todos os Anéis de Polinómios necessários para o restante do programa:**
   - Criação do parâmetro q - que corresponde a um número primo
   - Criação do parâmetro p - que corresponde também a um número primo
   - O parâmetro w é criado e usado em toda a parte da *Key Generation*, *Encapsulation* e *Desencapsulation*
   - Verificar se $x^p - x - 1$ é irredutível no Anel de Polinómios $(\mathbb{Z}/q)[x]$
   - Abreviar os 2 Anéis de Polinónimo $\mathbb{Z}[x]/(x^p - x - 1)$ e $(\mathbb{Z}/3)[x]/(x^p - x - 1)$ e o Campo/Grupo Finito $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ respetivamente como R, $R/3$ e R/q

   Note-se que o *field* criado é usado para verificar a irredutibilidade de $x^p - x - 1$, justificando-se assim a ordem a nível do código.

2. ***Key Generation* - Geração das Chaves (*Secret* e *Public Key*):**
   - Criação de um elemento pequeno g uniforme e aleatório tal que $g \in R$. Repetir o processo de modo a verificar que g é invertível em $R/3$

- Criação de um elemento pequeno f uniforme e aleatório tal que $f \in R$. Este elemento f deve ser de peso w, diferente de 0 e por isso, invertível em R/q
- O Segredo guardado será o **par ordenado** $(f, 1/g)$ em que f em R e $1/g$ em $R/3$
- A Chave Pública corresponderá ao cálculo $h = g/(3f)$ em R/q

3. ***Encapsulation* - Processo de Encapsulamento:**

   - Obtenção do $h \in R/q$. Para este teste algorítmico corresponderá à *Public Key* dada como parâmetro da função criada para o efeito
   - Criação de um elemento pequeno e uniforme e aleatório tal que $r \in R$. Este elemento r deve ser de peso w
   - Calcular $hr \in R/q$
   - Arredondar cada coeficiente de hr visto como um inteiro entre $-(q-1)/2$ e $(q-1)/2$, para o múltiplo de 3 mais próximo, obtendo-se assim $C \in R$

   A parte da Hash é feita na secção de teste criada para esta classe Python, de modo a se conseguir testar as versões PKE-IND-CCA e KEM-IND-CPA.

4. ***Decapsulation* - Processo de Desencapsulamento:**

   - Obtenção do valor de C que é passado como argumento
   - Multiplicar o valor de C por $3f$ em R/q
   - Arredondar cada coeficiente de $3fC$ visto como um inteiro entre $-(q-1)/2$ e $(q-1)/2$, reduzindo ao módulo 3, obtendo-se um polinómio em $R/3$
   - Multiplicar esse valor por $1/g \in R/3$
   - Calcular r'

1.3 Resolução do Exercício

1. *Parameter Space*

Geração dos Parâmetros e dos Anéis de Polinómios necessários para o restante do programa. Dado a necessidade de usar estas variáveis ao longo do algoritmo, criou-se uma célula à parte para permitir a globalidade do programa de forma correta.

```
In [1]: # Parâmetro primo q
        q = 24*64

        # Verifica se o valor dado ao q é primo
        # Cria-se um ciclo while para fazer essa verificação
        while True:

            if (1 + q).is_prime():
                break
            else:
```

```
        q = q + 3
    q = q + 1

    # Anéis de Polinómios e Campo/Grupo Finito
    Zx.<x> = ZZ[]                       # Anel de Inteiros
    Z3.<y> = PolynomialRing(GF(3))      # Anel do Grupo Finito do módulo 3
    Gq.<z> = GF(q)[]                    # Grupo Finito do módulo q

    # Parâmetro primo p
    # Dado que se declarou o primo q anteriormente pode-se fazer logo
↪uso da função next_prime
    p = next_prime(2*64)

    # Verifica se x^p - x - 1 é irredutível no Grupo Finito Gq
    # Cria-se um ciclo while para fazer essa verificação
    while True:

        if  Gq(x^p-x-1).is_irreducible():
            break
        else:
            p = next_prime(p+1)

    # Resolver/Abreviar os Anéis e o Campo/Grupo como R, R/3 e R/q
    ZxR.<x> = Zx.quotient(x^p-x-1)
    ZR3.<y> = Z3.quotient(y^p-y-1)
    GRq.<z> = Gq.quotient(z^p-z-1)
```

2, 3 e 4. ***Key Generation, Encapsulation* e *Decapsulation***

```
In [6]: # Imports Necessários
        import hashlib
        from random import choice, randint

        # Função auxiliar que gera um Pequeno Elemento Aleatório
        def smallPoly(p, t = None):

            if not t:
                return Zx([choice([-1,0,1]) for k in range(p)])

            u = floor(2*(p-1)//t)
            k = randint(0, u)
            l = [0]*p

            while k < p:
```

```python
            l[k] = choice([-1, 1])
            k += randint(1, u)

        return Zx(l)

    # Função auxiliar que trata de arredondar cada coeficiente visto␣
↪como um inteiro entre -(q-1)/2 e (q-1)/2
    # Função adaptada para funcionar tanto para o Encapsulamento como␣
↪Desencapsulamento
    def myRound(round3OrNot, hr = None, w = None, q = q):

        # Caso seja 0 significa que estamos a tratar do arredondamento␣
↪pedido pelo Desencapsulamento
        if round3OrNot == 0:
            r = q//2

            return Zx(map(lambda x: lift(x + r) - r, w.list()))

        # Caso contrário, pelo Encapsulamento
        # Daí o arredondamento para múltiplo de 3 mais próximo
        elif round3OrNot == 1:

            # Aredondar para multpilo de 3 mais próximo
            def mul3(x):
                return ((x/3).round())*3

            r = q//2

            return  Zx(map(lambda x: mul3(lift(x+r) - r), hr.list()))

    # Classe NTRU-Prime que trata da Geração das Chaves, Encapsulamento␣
↪e Desencapsulamento
    class NTRU_Prime:

        # Função de Hash
        def Hash(self, w):
            ww = reduce(lambda x,y: x + y.binary(), w.list(), "")

            return hashlib.sha256(ww).hexdigest()

        # Geração das Chaves (Segredo e Public Key)
        def keyGeneration(self):
```

```
        # Pequeno Elemento aleatório g pertecente a R
        # Verificar que g seja invertível em R/3
        g = smallPoly(p)

        # Daí se recorrer ao Anel de Polinómios (Z/3)[x]
        while not ZR3(g).is_unit():
            g = smallPoly(p)

        # Pequeno Elemento aleatório f pertencente a R de peso w
        # Verificar que f é diferente de 0 e invertível em R/q
        w = smallPoly(p, 64)
        f = GRq(w)

        # Calcular 1/g pertencente a R/3
        gInv = ZR3(g)^(-1)

        # Chave Privada (Secrets)
        # Par Ordenado (f, 1/g)
        secret = (f, gInv)

        # Chave Pública em Gqr (h)
        publicKey = GRq(g)/GRq(3*f)

        return (secret, publicKey)

    # Encapsulamento
    def encapsulate(self, publicKey):

        # Public Key h
        # Neste caso é o argumento passado como pk
        h = publicKey

        # Pequeno Elemento aleatório f pertencente a R de peso w
        w = smallPoly(p, 64)
        r = GRq(w)

        # Calcular hr
        # Arredondar cada coeficiente de hr
        C = myRound(1, h*r)

        return (w, C)

    # Desencapsulamento
    def decapsulate(self, secret, C):
```

```
        # Obter o valor de f e 1/g
        (f, gInv) = secret

        # Multiplicar C por 3f em R/q
        # Arredondar cada coeficiente de 3fc
        # Multiplicar por 1/g (gInv)
        e = gInv * ZR3(myRound(0, None, GRq(3*f) * GRq(C)))

        # Calcular r'
        rlinha = myRound(0, None, e, 3)

        return rlinha
```

1.4 Teste da Classe

Teste e Verificação para o modo PKE-IND-CCA

```
In [22]: # Criação da Instância
         ntruPrime = NTRU_Prime()

         # Geração das Chaves (Segredo e Chave Pública)
         (secret, publicKey) = ntruPrime.keyGeneration()

         # Cifrar
         (key,C) = ntruPrime.encapsulate(publicKey)

         # Decifrar e Verificar
         if key == ntruPrime.decapsulate(secret, C):
             print "As chaves correspondem."
             print(key)
         else:
             print "Erro na correspondência das chaves."
```

```
As chaves correspondem.
-x^246 + x^243 + x^242 + x^240 - x^239 - x^235 - x^229 - x^224 + x^217 +␣
 ↪x^213 - x^206 + x^203 + x^198 - x^196 - x^195 - x^194 + x^189 + x^186 +␣
 ↪x^181 - x^180 + x^175 + x^170 - x^166 - x^163 - x^159 + x^156 - x^151 +␣
 ↪x^148 + x^143 + x^140 + x^137 + x^133 - x^129 + x^123 - x^117 - x^116 -␣
 ↪x^109 + x^108 + x^105 - x^104 + x^97 + x^90 + x^84 + x^83 + x^78 - x^76␣
 ↪- x^75 - x^73 + x^66 - x^60 + x^54 - x^52 - x^46 + x^39 + x^38 - x^31 +␣
 ↪x^30 + x^25 - x^21 - x^19 + x^18 - x^17 + x^14 + x^8 - x^6 + x^5 + x^2 +␣
 ↪x
```

Teste e Verficação da versão KEM-IND-CPA

KEM é a randomização duma chave do tamanho correto do algoritmo utilizado (neste caso o NTRU-Prime) e depois a chave simétrica a utilizar futuramente, na verdade está encapsulada utilizando um KDF (ex: SHA-256). Assim esse KDF irá derivar a chave "grande"do PKE para uma chave simétrica com o tamanho correto (SHA-256 dá valores de 256-bits, que podem ser utilizados num algoritmo simétrico como AES-256)

In [23]:
```
# Criação da Instância
ntruPrime = NTRU_Prime()

# Geração das Chaves (Segredo e Chave Pública)
(secret, publicKey) = ntruPrime.keyGeneration()

# 1. Bob

# Cifrar
(keyBob, C) = ntruPrime.encapsulate(publicKey)

# Passar a Chave de tamanho grande do PKE NTRU-Prime para 256 bits␣
↪usando SHA-256
chaveSimetricaBob = ntruPrime.Hash(keyBob)

# 2. Alice

# Cifrar
keyAlice = ntruPrime.decapsulate(secret, C)

# Se as chaves PKE coincidirem (Algoritmo funciona corretamente)
if keyBob == keyAlice:
    chaveSimetricaAlice = ntruPrime.Hash(keyAlice)
else:
    print "O Algoritmo NTRU não decifrou corretamente."

# Verificar a validade de ambas as Chaves Simétricas
if chaveSimetricaBob == chaveSimetricaAlice:
    print "As chaves correspondem."
    print(chaveSimetricaBob)
else:
    print "Erro na correspondência das chaves."
```

```
As chaves correspondem.
74c63fbed90859b8d5198d8ec717686a655ed1e75ad84bce791f56d76ff03497
```

1.5 Observações Finais

- O algoritmo **NTRU** ajudou a ter uma noção muito basilar acerca do funcionamento do **NTRU-Prime**;
- Através do documento de envio principal do **NTRU-Prime** conseguiu-se compreender bem todos os passos e com isso criar as várias etapas que formam o algoritmo, permitindo testar toda a validade do programa Python/Sagemath;
- A grande dificuldade continua a ser, tal como nos outros Trabalhos Práticos, toda a ideia de trabalhar com o Sagemath e compreender todas as suas funcionalidades necessárias para os vários algoritmos criptográficos.

1.6 Referências

- ENS DE LYON, NTRU Prime: Intro `https://ntruprime.cr.yp.to` (Acedido a 10 maio 2020)
- Wikipedia, NTRU `https://en.wikipedia.org/wiki/NTRU` (Acedido a 11 maio 2020)

2 Implementação do NewHope com o *SageMath*

2.1 Descrição do Exercício

Construir uma classe Python/SageMath que implemente o esquema KEM **NewHope** a concurso do NIST-PQC.

2.2 Descrição da Implementação

Para a implementação houve a criação de um campo ciclotómico, através dum anel polinomial dum grupo finito de 1024, logo ficando $R/[X^1024+1]$.

1. **Para a inicialização do algoritmo NewHope fez-se:**
    - Uma matriz identidade de sub-dimensão 4
    - Uma rede diagonal de inteiros a partir dessa matriz
    - Criar um meio vetor e modificar a última linha da matriz com ele
    - Criação do poliedro principal com as estruturas matemáticas anteriores
2. **Para a geração de noise/erro:**
    - Buscamos um *sample* polinomial da Distribuição Gaussiana
3. **Para a geração do sinal:**
    - Vamos fazer a divisão dos elementos pelo módulo tendo os coeficientes
    - Caso o vetor V esteja no poliedro principal, utiliza-se. Senão vai-se à rede diagonal e usa-se o vetor mais próximo de V.
    - Sendo assim a distância, o sinal.
4. **Para a reconciliação é um processo análogo:**
    - Vamos fazer a divisão dos elementos pelo módulo tendo os coeficientes
    - Adicionamos 1 caso a coordenada esteja no centro do poliedro, senão 0
    - Temos assim uma chave.

2.3 Resolução do Exercício

```
In [1]: import itertools, numpy

        from sage.stats.distributions.discrete_gaussian_polynomial import␣
 ↪DiscreteGaussianDistributionPolynomialSampler
        from sage.modules.free_module_integer import IntegerLattice
        from sage.modules.diamond_cutting import calculate_voronoi_cell
```

```
    dimension = 1024     # Grau dos Polinómios (Dimensão suportada pelo␣
↪NewHope)
    modulus = 12289      # Módulo (Valor de q)
    sigma = 8/sqrt(2*pi) # Sigma (Com o Parâmetro de Distribuição de␣
↪Ruído = 8)

    # Anel Polinomial Quociente
    R.<X> = PolynomialRing(GF(modulus))      # Anel polinomial
    Y.<x> = R.quotient(X^(dimension) + 1)    # Campo Ciclotómico

    subDimension = 4

    # Função Auxiliar que converte do Intervalo [c0, c1, ..., c1023]␣
↪para [(c0, c1, c2, c3), ..., (..., c1023)]
    def grouped(iterable, n):
        return zip(*[iter(iterable)]*n)

    class NewHope:

        # Inicialização Algortimo
        def initialize(self):

            # Criação da Matriz Identidade
            # Construir uma Rede diagonal de Inteiros a partir da␣
↪Matriz Identidade
            identityMatrix = Matrix.identity(RR, subDimension)
            integerLattice = IntegerLattice(identityMatrix)

            # Construir um Half Vector (1/2)
            # Modificar última linha da Matriz Identidade
            halfVector = [1/2 for i in range(subDimension)]
            identityMatrix[subDimension - 1] = halfVector

            # Criar Célula Voronoi a partir Matriz Modificada
            mainPolyhedron = calculate_voronoi_cell(identityMatrix).
↪translation(halfVector)

            # Retorna uma Lattice de Inteiros e um Poliedro de centro␣
↪em (1/2, ..., 1/2)
            return (integerLattice, mainPolyhedron)

        def dbl(self, coefficient_vector):
            return  coefficient_vector + \
```

```
                      vector( numpy.random.choice([0, 1], p=[0.5, 0.5]) *
vector([1/(2*modulus) for _ in range(subDimension)]))

        # Geração do Erro
        def generateError(self):
            f = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'],
5, sigma)()
            return Y(f)

        # Geração do Polinómio a partir de Elementos Random
        def generatePolynomial(self):
            return Y.random_element()

        # Geração do Signal
        def generateSignal(self, poly):

            # Divide coeficientes pelo módulo
            coefficients = map(lambda x: RR(x) / modulus, poly.list())
            distances = []

            for v in grouped(coefficients, subDimension):

                v = self.dbl(vector(v))

                # Caso o ponto/vetor esteja no  Main Polyhedron, usa-se
o centro do Main Polyhedron
                if mainPolyhedron.contains(vector(v)):
                    distance = mainPolyhedron.center() - v
                # Caso contrário
                else:
                    distance = integerLattice.closest_vector(v) - v
                distances.append(distance)

            return distances

        # Reconciliação
        def reconcile(self, poly, w):

            coefficients = map(lambda x: RR(x) / modulus, poly.list())
            key = []

            for difference, v in zip(w, grouped(coefficients,
subDimension)):
```

```
                    v = self.dbl(vector(v))
                    coordinate = vector([round(point, 1) for point in (v +␣
↪difference) ])
                    key.append(1 if coordinate == mainPolyhedron.center()␣
↪else 0)

                return "".join(map(str, key))
```

2.4 Teste da Classe

Para teste da classe e algoritmo total fez-se:

- Inicialização do anel diagonal e o poliedro
- Cria-se uma matriz partilhada para os dois
- Instanciação dos valores do Bob (segredo, erro e o valor)
- Instanciação dos valores da Alice (segredo, erro e o valor)
- A chave da Alice sendo: $ valueBob * secretAlice + temp_error$
- A chave do Bob sendo: $ valueAlice * secretBob $
- Fazer a reconciliação destes polinómios e ver se são iguais.

Teste e Verificação para o modo PKE-IND-CCA

```
In [2]: # Criação da Instância
        newHope = NewHope()

        # Lattice de Inteiros e Poliedro Principal
        (integerLattice, mainPolyhedron) = newHope.initialize()

        # Matriz Partilhada
        shared = newHope.generatePolynomial()

        # 1. Valores Bob
        secretBob = newHope.generateError()
        errorBob = newHope.generateError()
        valueBob = shared * secretBob + errorBob

        # 2. Valores Alice
        secretAlice = newHope.generateError()
        errorAlice = newHope.generateError()
        valueAlice = shared * secretAlice + errorAlice

        # Key Alice
        temp_error = newHope.generateError()
        keyAlice = valueBob * secretAlice + temp_error
```

```
    w = newHope.generateSignal(keyAlice)
    keyBinaryAlice = newHope.reconcile(keyAlice, w)

    # Key Bob
    keyBob = valueAlice * secretBob
    keyBinaryBob = newHope.reconcile(keyBob, w)

    if (keyBinaryBob == keyBinaryAlice):
        print "As chaves correspondem."
        print hex(int(keyBinaryBob, 2))
    else:
        print "Erro na correspondência das chaves."
```

```
As chaves correspondem.
0x6407f9782c69ab7ee51a61a4235109af87ee0d0f40cc1d9e8899e1c0050e2d0L
```

2.5 Observações Finais

- O algoritmo **NewHope** foi muito complicado de entender, dado que o próprio documento principal de envio não continha o algoritmo propriamente descrito, tal como acontecia para o **NTRU-Prime**;
- Recorreu-se à implementação deste algoritmo realizado através de outras linguagens de programação (incluindo o próprio Python), de forma a tentar compreender todo o processo algorítmico por detrás.

2.6 Referências

- GitHub, New Hope Key Exchange and Key Encapsulation https://github.com/Art3misOne/newhope (Acedido a 10 maio 2020)
- GitHub, PyNewHope https://github.com/scottwn/PyNewHope (Acedido a 10 maio 2020)