MIEI/MI - Estruturas Criptográficas

Trabalho Prático 3

João Alves
a77070@alunos.uminho.pt

Nuno Leite
a70132@alunos.uminho.pt

Universidade do Minho
8 de Maio de 2019

1 Introdução

A resolução deste trabalho prático tem 3 objetivos principais:

- Criar uma classe **Python** que implemente o algoritmo de *Boneh & Venkatesan* .
- Implementação de um esquema de assinaturas digitais *NTRUEncrypt* .
- Estudar gamas de valores de determinados parâmetros que tornem viável um ataque de inversão de chave pública ou inversão do criptograma utilizando redução de bases.

Este relatório está, desta forma, dividido em três partes, cada parte correspondente à resolução de um dos problemas e, além disso, está estruturado de forma a que o texto entre os *snippets* de código seja suficientemente explicativo sobre a implementação e desenho da solução em cada um dos problemas.

2 Algoritmo de Boneh & Venkatesan

Esta secção tem como propósito definir uma classe **Python** que implemente o algoritmo de *Boneh & Venkatesan* , que explora o *Hidden Number Problem* que, se bem sucedido, permite obter um segredo a partir de um conjunto de dados. Além disso, a secção está dividida em quatro partes:

- A primeira parte define o oráculo HNP, que gera um segredo,permite calcular o msb e compara um segredo calculado com o gerado;
- A segunda parte define as funções de geração do vetor aleatório **x** e do vetor composto por $ui = msb(xi, s)$ para $i = 0$ até $i = l - 1$.
- A terceira parte define apenas o algoritmo de *Boneh & Venkatesan* .
- A quarta parte, apelidada de teste, fornece os parâmetros necessários à instância do algoritmo para que ele descubra o segredo **s** a partir dos mesmos.

2.1 Definição do oráculo HNP

O propósito desta secção passa por definir o oráculo que gera um segredo e, posteriormente, retorna o vetor u, com l elementos, onde cada um é um inteiro representativo dos k bits mais significativos de $s * xi$.

```
In [52]: class HNPOracle:

             def __init__(self,p):
                 # gerar o segredo.
                 self.secret = ZZ.random_element(p)
                 print 'secret'
                 print self.secret

             def msb(self,k,p,xi):
                 value = ZZ(Mod(xi*self.secret,p))
                 binary_value = value.digits(2)
                 binary_value.reverse()
                 value_str = ''
                 if len(binary_value) < k:
                     for i in range(0,len(binary_value)):
                         value_str += str(binary_value[i])
                 else:
                     for i in range(0,k):
                         value_str += str(binary_value[i])
                 return value_str

             def compare_secret(self,calculated_secret):
                 if calculated_secret == self.secret:
                     return True
                 else:
                     return False
```

2.2 Definição da função de geração do vetor aletório e vetor u

```
In [53]: def generate_l_random_elements(l,p):
             x = []
             for i in range(0,l):
                 x.append(ZZ.random_element(p))
             return x

         def calculate_u_vector(x_vector,k,p,oracle):
             u_vector = []
             for xi in x_vector:
                 ui = oracle.msb(k,p,xi)
                 ui = ZZ(int(ui,2))
                 u_vector.append(ui)
             return u_vector
```

2.3 Definição do algoritmo de Boneh & Venkatesan

```
In [54]: import sage.modules.free_module_integer as fmi
         import numpy as np
```

```
class BV:

    def __init__(self,u,x,k,p,l):
        # construir a matriz L , lambda,o target T e, finalmente, o reticulado Lret
        self.param_lambda = 2^(k+1)
        self.L = self.param_lambda * p * matrix.identity(l)
        self.L = self.L.transpose()
        self.L = self.L.insert_row(l,zero_vector(l))
        self.L = self.L.transpose()
        temp_x = [self.param_lambda * i for i in x]
        temp_x.append(1)
        self.L = self.L.insert_row(l,temp_x)
        self.target = [self.param_lambda * i for i in u]
        self.target.append(0)
        self.target = matrix(self.target)
        self.Lret = fmi.IntegerLattice(self.L)


    def solve(self,x,p,l):
        # Calcular o CVP aproximado do reticulado Lret
        L = matrix(self.Lret.reduced_basis)
        t = matrix(1,l+1,list(-self.target))
        zero = matrix(l+1,1,[0]*(l+1))
        M = matrix(1,1,p**2)
        L1 = block_matrix(2,2,[[L,zero],[t,M]])
        ret = fmi.IntegerLattice(L1).reduced_basis
        error1 = np.array(ret[l+1][:-1])
        y1 = error1 + self.target
        return y1[0][l] # última componente do vetor resultante.
```

2.4 Teste ao algoritmo de Boneh & Venkatesan

In [78]:
```
l = 2^7
k = 64
p = 2^64
x_vector = generate_l_random_elements(l,p)
oracle = HNPOracle(p)
u_vector = calculate_u_vector(x_vector,k,p,oracle)
bv = BV(u_vector,x_vector,k,p,l)
calculated_secret = bv.solve(x_vector,p,l)
print 'calculated_secret'
print calculated_secret
if oracle.compare_secret(calculated_secret):
    print 'Algoritmo aproximado calculou o segredo com sucesso!'
else:
    print 'Algoritmo aproximado não conseguiu calcular o segredo com sucesso!'
```

```
secret
6981224216842594255
calculated_secret
6981224216842594255
Algoritmo aproximado calculou o segredo com sucesso!
```

3 Definição do Esquema NTRU-Encrypt

Nesta secção foi definido um esquema criptográfico, com base na cifra NTRU disponibilizada num notebook Sage, o artigo de Joseph Silverman e a documentação das candidaturas NTRU-Encrypt e NTRU-Prime apresentadas ao concurso *NIST* de standards *PQC*, de uma destas.

A implementação aqui documentada, refere-se ao esquema criptográfico definido como *ntru-pke*, que consiste em criptografia de chave pública.

3.1 Módulos Importados e Parâmetros Públicos

Antes da definição dos algoritmos, é necessário definir os seguintes parâmetros públicos e importar os módulos necessários.

```
In [183]: import random
          import hashlib
          from datetime import datetime
          from sage.crypto.util import ascii_to_bin, bin_to_ascii

          name        = "NTRU_PKE_443"
          d           = 115
          N           = 443
          p           = 3
          q           = next_prime(p*N)
          max_msg_len = 33
          Z.<x>       = ZZ[]
          Q.<x>       = PolynomialRing(GF(q),name='x').quotient(x^N-1)
```

O parâmetro `q` foi definido pela primitiva `next_prime(p*N)`, com base no notebook Sage, embora a candidatura tenha usado 2048, para o valor deste. Isto deveu-se ao facto do método `lift` resultar na seguinte mensagem de erro, aquando do uso desse valor para o parâmetro `q`.

———————————————————————————————————————

/usr/lib/python2.7/site-packages/sage/misc/functional.pyc in lift(x)

972 return x.lift()

973 except AttributeError:

–> 974 raise ArithmeticError("no lift defined.") ArithmeticError: no lift defined.

3.2 Definição de Funções Auxiliares

Feito isto, definiram-se algumas funções auxiliares, que facilitaram a implementação dos algoritmos descritos na próxima secção.

```
In [184]: def pad(msg):
              msg_len = len(msg)/8
              if msg_len > max_msg_len:
                  raise Exception('msg_len should not exceed {}. The value of msg_len was: {}'
                                  .format(max_msg_len, msg_len))

              r = N - (167 + 6 + msg_len*8)
              lr = list(0 for i in (0..r-1))

              msg_len = "{0:06b}".format(int(msg_len))
              msg_len = [int(d) for d in msg_len[:6]];

              m = msg + lr + vec(167) + msg_len;

              return m

          def hash_message(m, h):
              m = ''.join([str(x) for x in m])
              hm = hashlib.sha512(m)
              lh = map(lift,h.list())
              sh = ''.join(str(x) for x in lh)
              hh = hashlib.sha512(sh)
              rseed = str(hm) + str(hh)
              return rseed

          def vec(n):
              return  [choice([-1,0,1]) for k in range(n)]

          # arredondamento módulo 'q'
          def qrnd(f):
              qq = (q-1)//2 ; ll = map(lift,f.list())
              return [n if n <= qq else n - q  for n in ll]

          # arredondamento módulo 'p'
          def prnd(l):
              pp = (p-1)//2
              rr = lambda x: x if x <= pp else x - p
              return [rr(n%p) if n>=0 else -rr((-n)%p) for n in l]

          def extract(m):
              msg_len = m[-6:]
              msg_len = ''.join(str(x) for x in msg_len)
              msg_len = int(msg_len,2)

              msg = m[:msg_len*8]

              return msg, msg_len
```

3.3 Definição do Esquema

De seguida, num sistema criptográfico **NTRU**, `f` é a chave privada e `h` a chave pública. Estas são definidas pelo seguinte algoritmo e implementadas no método `keypair`.

NTRU.keypair
Input: *seed*

1. Instanciar um *Sampler* com uma *seed*
2. f <- *Sampler*
3. Se f não for invertível $\mod p$, voltar ao passo 2
4. g <- *Sampler*
5. $h = g/(pf+1) \mod q$

Ouput: Chave pública h e chave privada (pf, g)

NTRU.encrypt
Input: Mensagem *msg* (representada sob a forma de uma lista de bits)

1. $m = pad(msg)$
2. $rseed = hash(m|h)$
3. Instanciar um *Sampler* com *rseed*
4. r <- *Sampler*
5. $t = r \times h$
6. $tseed = hash(m|h)$
7. Instanciar um *Sampler* com *tseed*
8. m_{mask} <- *Sampler*
9. $m' = m - m_{mask}$
10. $c = t + m'$

Output: Texto cifrado c

Por último, definiu-se o algoritmo para decifrar o texto, implementado no método `decrypt`, da seguinte forma:

NTRU.decrypt
Input: Texto cifrado *c*

1. $m' = f \times c \mod p$
2. $t = c - m$
3. $tseed = hash(t)$
4. Instanciar um *Sampler* com *tseed*
5. m_{mask} <- *Sampler*
6. $m = m' + m_{mask} \mod p$
7. $rseed = hash(m|h)$

8. Instanciar um *Sampler* com *rseed*
9. *r* <- *Sampler*
10. $msg, msg\_len = extract(m)$

Output: *msg*

```
In [185]: class NTRU:

              def keypair(self, seed):
                  f = Q(0)
                  random.seed(seed)
                  while not f.is_unit():
                      F = Q(vec(N)); f = 1 + p*F
                  G = Q([choice([-1,0,1]) for k in range(N)]) ; g = p*G
                  self.f = f
                  self.h = f^(-1) * g

              def encrypt(self, msg):
                  m = pad(msg)

                  rseed = hash_message(m, self.h)
                  random.seed(rseed)
                  r = [random.choice([-1,0,1]) for k in range(N)]

                  t = Q(r) * self.h

                  lt = map(lift,t.list())
                  st = ''.join(str(x) for x in lt)
                  tseed = hashlib.sha512(st)
                  random.seed(tseed)

                  m_mask = [random.choice([-1,0,1]) for k in range(N)]
                  mm = prnd(qrnd(Q(m) - Q(m_mask)))

                  c = t + Q(mm)

                  return c

              def decrypt(self,e):
                  mm = prnd(qrnd(self.f * e))
                  t = e - Q(mm)

                  lt = map(lift,t.list())
                  st = ''.join(str(x) for x in lt)
                  tseed = hashlib.sha512(st)

                  random.seed(tseed)
                  m_mask = [random.choice([-1,0,1]) for k in range(N)]
```

```
        m = prnd(qrnd(Q(mm) + Q(m_mask)))

        rseed = hash_message(m, self.h)
        random.seed(rseed)
        r = [random.choice([-1,0,1]) for k in range(N)]

        msg,msg_len = extract(m)

        return msg
```

3.4 Teste ao esquema NTRU

```
In [186]: K = NTRU()

          msg = vec(33 * 8)
          print('\nmessage:\t' + str(msg))
          print('message length:\t' + str(len(msg)/8) + ' bytes' + '\n')

          seed = datetime.now()
          K.keypair(seed)
          e = K.encrypt(msg)
          print msg == K.decrypt(e)


message:        [0, 0, 1, -1, -1, -1, 1, 0, 1, -1, -1, 1, 0, 1, 1, -1, 1, 0,
             -1, 1, -1, -1, -1, 0, -1, 1, 0, 0, 1, 0, 1, 1, -1, 1, 1, 0,
             0, 0, -1, -1, 0, -1, 0, -1, 1, -1, 0, 0, -1, -1, -1, -1, 1,
             -1, -1, 0, 0, 1, 0, 1, -1, 0, 0, -1, 0, 1, 0, 1, 1, 1, 1, 1,
             0, 0, 1, 0, -1, -1, -1, 1, 0, 1, 0, 1, 1, 1, 0, 1, -1, -1, 1,
             0, 1, 0, 1, -1, 0, 1, 0, -1, 0, -1, -1, 0, -1, -1, 1, 1, 0, 0,
             1, 1, 0, -1, 1, 0, 1, 1, -1, -1, 1, 0, -1, -1, 0, 1, 1, 0, 0,
             -1, 0, 1, 0, -1, 1, 1, 1, -1, 0, 0, -1, -1, -1, -1, 1, 1, 0,
             -1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, -1, 1, 1, 1, 0, 0,
             0, 0, -1, -1, 0, 0, 0, -1, -1, -1, 0, 0, 0, 0, 1, -1, 1, 1,
             -1, 0, -1, 1, 1, 0, 0, 0, 0, -1, 1, 0, -1, 0, 0, -1, 1, -1,
             1, 0, 1, 1, -1, 0, 1, 1, 1, -1, -1, -1, 1, -1, 0, 0, 1, -1,
             1, 0, 1, 0, -1, 1, 1, 0, -1, 1, 0, -1, 0, 1, 0, -1, 0, 1,
             -1, 1, -1, 0, -1, 0, 1, 0, -1, -1, -1, 1, -1, 1, 1, 0, 0,
             1, -1, 0, 1, 1, -1, 1, 0, -1]
message length:         33 bytes

True
```

4 Ataques de inversão

4.1 Definição da classe NTRU e da classe LAT

In [135]:
```
import sage.modules.free_module_integer as fmi

d = 6

N = 21
p = 3
q = next_prime(p*N)

Z.<x>  = ZZ[]         # polinómios de coeficientes inteiros
Q.<x>  = PolynomialRing(GF(q),name='x').quotient(x^N-1)

def vec():
    return  [choice([-1,0,1]) for k in range(N)]

# arredondamento módulo 'q'
def qrnd(f):    # argumento em 'Q'
    qq = (q-1)//2 ; ll = map(lift,f.list())
    return [n if n <= qq else n - q  for n in ll]

# arredondamento módulo 'p'
def prnd(l):
    pp = (p-1)//2
    rr = lambda x: x if x <= pp else x - p
    return [rr(n%p) if n>=0 else -rr((-n)%p) for n in l]

class NTRU(object):
    def __init__(self):
        # calcular um 'f' invertível
        f = Q(0)
        while not f.is_unit():
            F = Q(vec()); f = 1 + p*F
        # gerar as chaves
        G = Q(vec()) ; g = p*G
        self.f = f
        self.h = f^(-1) * g

    def encrypt(self,m):
        r = Q(vec())
        return r*self.h + Q(m)

    def decrypt(self,e):
        a = e*self.f
        return prnd(qrnd(a))
```

```
class Lat(NTRU):
    def __init__(self):
        super(Lat,self).__init__()
        B1 = identity_matrix(ZZ,N); Bq = q*B1; B0 = matrix(ZZ,N,N,[0]*(N^2))
        h = qrnd(self.h)
        # rodar um vetor
        H = [h]
        for k in range(N-1):
            h = [h[-1]] + h[:-1]    # shift right rotate
            H = H + [h]
        H = matrix(ZZ,N,N,H)
        self.L = fmi.IntegerLattice(block_matrix([[Bq,B0],[H,B1]]))
```

4.2 Inversão da chave pública

Segundo o artigo de *Silverman*, encontrar a chave privada f a partir da chave pública h, é equivalente a encontrar um vetor curto no reticulado **L(h)** definido em cima, para parâmetros apropriados.

```
In [178]: import numpy as np

          l = Lat()
          lredmat = l.L.reduced_basis.LLL()
          lred = fmi.IntegerLattice(lredmat)

          short_aproximate = np.array(lredmat[0][:-1]) #SVP aproximado
          print Q(prnd(short_aproximate))
          print l.f
```

```
x^20 + x^16 + 66*x^13 + x^11 + x^10 + 66*x^8 + 66*x^7 + x^6 + x^4 + x^3 + 66*x + 2
3*x^19 + 64*x^17 + 3*x^16 + 3*x^13 + 3*x^11 + 3*x^10 + 3*x^9 + 3*x^8 + 3*x^7 + 64*x + 1
```

4.3 Inversão do criptograma

Segundo o artigo de *Silverman*, recuperar a mensagem original a partir do criptograma e da chave pública, é equivalente a encontrar o vetor mais próximo do target $[0, criptograma]$, no reticulado **L(h)**.

```
In [174]: import numpy as np

          l = Lat()
          message = vec()
          print 'message:'
          print Q(message)
          e = l.encrypt(message)
          e = qrnd(e)
          vector = [0 for i in range(0,N)]
          for i in e:
```

```
            vector.append(i)
        zero_41_vector =  [0 for i in range(0,41)]
        zero_41_vector.append(2**q)
        lred = l.L.reduced_basis
        lred = lred.transpose()
        lred = lred.insert_row(42,zero_41_vector)
        lred = lred.transpose()
        L1 = fmi.IntegerLattice(lred)
        lred = L1.reduced_basis

        err1 = np.array(lred[41][:-1])
        y1 = err1 + vector
        # y1 deverá ser igual ao vetor [-r,m], retirámos -r e ficámos apenas com m.
        new_vec = []
        for i in range(22,42):
            new_vec.append(y1[i])
        print ''
        print 'calculated_message'
        print Q(prnd(new_vec))

message:
66*x^19 + 66*x^18 + 66*x^17 + 66*x^15 + 66*x^14 + x^12
     + 66*x^11 + x^10 + 66*x^9 + x^7 + 66*x^6 + x^3
     + x^2 + x + 1

calculated_message
66*x^19 + x^18 + x^15 + 66*x^14 + x^13 + 66*x^12
     + 66*x^11 + x^10 + x^9 + x^6 + x^4 + x^3
     + x^2 + 66*x + 1
```

5 Conclusão

Os resultados da realização deste trabalho prático não são, desta vez, tão satisfatórios como nos trabalhos anteriores visto que, apesar de termos conseguido implementar o algoritmo de **Boneh & Venkatesan** e o esquema **NTRUEncrypt**, não conseguimos implementar, de forma totalmente correta, os ataques de inversão referidos no enunciado. Como é dito em cima, conseguimos entender a relação entre o cálculo do vector mais curto e o cálculo do vetor mais próximo com as inversões da chave pública e criptograma, respetivamente, mas não conseguimos implementar de forma a que conseguíssemos obter esses mesmos resultados exatos.

Como deve ser óbvio nesta altura, este trabalho apresentou imensas dificuldades, especialmente e decididamente na pergunta 3, mas também no esquema **NTRUEncrypt**, apesar de que, nesse caso, ainda o conseguimos implementar de forma a que consiga cifrar e decifrar, corretamente, uma mensagem.

6 Referências

1. Worksheets TP4 do professor
2. NTRU and Lattice-Based Crypto: Past, Present, and Future de Joseph H. Silverman
3. NTRUEncrypt Supporting Documentation
4. HNP e abordagem de Boneh & Venkatesan