Estruturas criptograficas: TP3 problema 1

Introdução

Nguyen & Shparlinsk, propõem reduções do Hidden Number Problem (HNP) a problemas difíceis em reticulados. Neste trabalho pretende-se construir, com a ajuda do Sagemath, uma implementação da solução discutida nos apontamentos para resolver o HNP com soluções aproximadas dos problemas em reticulados.

Descrição

Para tal foi criada uma classe com os valores iniciais, **p**, **k** (calc_k), **n** (calc_n), **lam**, onde também é gerado um segredo pseudo-aletório **s**, e os vetores **xs** e **us**, a partir de s.

O passo seguinte consistiu em criar a matriz **G**, contruída a partir da matriz identidade (de tamanho n) multiplicada por **p**, onde à mesma foram adicionadas duas colunas de zeros, e duas linhas, onde a penúltima é o vetor **xs**, seguido de [A,0], e a última linha é contituída pelo vetor **us** * **-B**, seguido de [0,M]. De seguida, é aplicada a redução de base Lattice, à matriz **G**.

Teste

De modo a testar o **HNP**, foi implementada a função **compare**, que calcula o segredo a partir do penúltimo elemento da última linha, e compara-o com o segredo **s**.

In [1]:
```
class HNP():
    def __init__(self, p):
        self.p = p
        self.k = self.calc_k(p)
        self.n = self.calc_n(p)
        self.lam = 2^self.k

        # Problem parameters

        # determinar  um segredo s != 0 em Zp
        self.s = ZZ.random_element(1,self.p)

        # a partir de uma sequência de n pares (x_i, u_i) em Zp x Zp
        self.xs = [ZZ.random_element(1,self.p) for i in range(self.n)]
        self.us = [self.msb(self.s*x % self.p) for x in self.xs ]


    def calc_k(self, p):
        d = log(p,2)
        k = ceil(sqrt(d)) + ceil(log(d,2))
        return k
```

```
    def calc_n(self, p):
        d = log(p,2)
        n = 2*ceil(sqrt(log(p,2)))
        return n

    # most significant bits
    def msb(self, y):
        B = self.p / self.lam
        return floor(y / B)

    def reduction_matrix(self):

        A = 1 / self.lam
        B = self.p / self.lam
        m = self.n + 2
        M = self.p * self.lam

        # construção da matriz G

        # penúltima linha
        lineX = Matrix(QQ, 1, m, [x for x in self.xs] + [A,0])
        # última linha
        lineU = Matrix(QQ, 1, m, [-u * B for u in self.us] + [0,M])
        # últimas duas colunas a zeros
        zeros = Matrix(QQ, self.n, 1, [0] * self.n)
        # matriz identidade de dimensão n*n
        ident = p * identity_matrix(ZZ, self.n)
        # junção da identidade com as colunas a zeros
        init = block_matrix(QQ, 1, 3,[ident, zeros, zeros])
        # junção de todas as linhas
        self.G = block_matrix(QQ, 3, 1, [init, lineX, lineU])

        # algoritmo de redução de base Lattice
        self.G = self.G.LLL()


    def compare(self):

        # penúltimo elemento da última linha
        e   = self.G[-1][-2]

        # determina-se o segredo s como |lam * e_n+1|
        calc_s = floor(e * self.lam) % self.p

        if self.s == calc_s:
            print("Correct secret!\n")
            print("Generated secret: \t", self.s)
            print ("Calculated secret: \t", calc_s)
        else:
            print("Incorrect secret")
```

In [2]:

```
bits = 512
p = random_prime(2^bits, lbound=2^(bits-1))

hnp = HNP(p)
hnp.reduction_matrix()
hnp.compare()
```

```
Correct secret!

Generated secret:        5207087788851119415225454835876191962677858304758406025782080391280632323382972524120955452945765312647403623555466417698850524745760426323430152435340794
Calculated secret:       5207087788851119415225454835876191962677858304758406025782080391280632323382972524120955452945765312647403623555466417698850524745760426323430152435340794
```