Estruturas Criptográficas

Trabalho Prático 3 - Exercício 1

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

No capítulo 5 dos apontamentos, é descrito o chamado **Hidden Number Problem**. No capítulo 8 dos apontamentos, é discutido um artigo de **Nguyen & Shparlinsk**, onde se propõem reduções do **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.

Resolução

Todo o código foi desenvolvido com base na íntegra dos apontamentos referidos no enunciado do problema. Desta forma, passa-se, então, a explicar as diversas funções utilizadas durante a realização do exercício proposto:

- **msb_k**: função que extrai, sob a forma de um inteiro positivo, os k *bits* mais significativos do argumento
- **generate_pairs**: algoritmo que gera os pares que obedecem à regra $u_i = msb_k(\lfloor s \times x_i \rfloor_p)$ para todo i = 1..n
- **build_lattice**: algoritmo que produz o reticulado, a partir da matriz geradora $G' \in Q^{m \times m}$, com m = n + 2, sendo n a dimensão dos pares gerados
- **reduce_lattice**: função responsável por aplicar a redução ao reticulado
- **find_secret**: algoritmo com a capacidade de recuperar o segredo

In [41]:
```
def msb_k(y, B):
    return y // B


def generate_pairs(n, p, s, B):
    pairs = []
    for _ in range(n):
        x_i = randint(0, p - 1)
        u_i = msb_k((s * x_i) % p, B)
        pairs.append((x_i, u_i))
```

```
    return pairs


def build_lattice(xs, us, A, n, p, lmbda, B):
    basis_vectors = []

    for i in range(n):
        vector = [0] * (n + 2)
        vector[i] = p
        basis_vectors.append(vector)

    basis_vectors.append(xs + [A] + [0])

    M = lmbda * p
    basis_vectors.append([-B * u for u in us] + [0] + [M])

    return Matrix(QQ, basis_vectors)


def reduce_lattice(lattice):
    return lattice.LLL()


def find_secret(reduced_lattice, lmbda, p):
    return (reduced_lattice[-1][-2] * lmbda).ceil() % p
```

Testes de aplicação

Para efeitos de teste, desenvolveu-se a função **solve_HNP** que, agregando todas as funções anteriormente descritas, resolve o problema **HNP**. Acrescentam-se, também, as regras às quais a função obedece de forma a que a resolução aconteça da forma esperada:

- **p** é um valor primo
- **k** é menor que $\log_2 p$
- **s** obedece à regra $s \neq 0 \in Z_p$
- **lmbda** obedece a $\lambda \equiv 2^k$
- **A** é da forma $A \equiv 1 / \lambda$
- **B** obedece a $B \equiv p / \lambda$

In [42]:
```
def solve_HNP(d):
    p = next_prime(2 ** d)
    k = d - 1
    lmbda = 2 ** k
    A = 1 / lmbda
    B = p / lmbda
    n = 2 * d
    s = randint(1, p - 1)
    print('s:', s)
```

```python
    pairs = generate_pairs(n, p, s, B)
    lattice = build_lattice([x for x, _ in pairs], [u for _, u in pairs], A,
    reduced_lattice = reduce_lattice(lattice)
    recovered_s = find_secret(reduced_lattice, lmbda, p)
    print('recovered s:', recovered_s)

    if (s == recovered_s):
        print('HNP solved!')

    else:
        print("Couldn't solve HNP!")
```

Apresentam-se, assim, três diferentes testes efetuados com diferentes valores de **d**.

In [43]:
```python
solve_HNP(16)
```

```
s: 31900
recovered s: 31900
HNP solved!
```

In [44]:
```python
solve_HNP(32)
```

```
s: 2294040602
recovered s: 2294040602
HNP solved!
```

In [45]:
```python
solve_HNP(64)
```

```
s: 12101661394542985602
recovered s: 12101661394542985602
HNP solved!
```