Estudo e Implementação do *Paillier Cryptosystem*

Universidade Estadual de Campinas

Felipe Santos Oliveira
RA 119383

30 de junho de 2015

1 Introdução

O criptosistema *Paillier* é um cripstosistema probabilístico de chave pública (ou assimétrico), cuja segurança se baseia na dificuldade da fatoração de números primos. Foi criado em 1999 pelo pesquisador Pascal Paillier, e possui características similares aos do RSA quanto à geração de chaves. No entanto, difere em seu aspecto probabilístico presente na operação de encriptação dos dados. O criptosistema de Paillier também possui propriedades homomórficas, discutidas na seção seguinte.

2 Descrição do Criptosistema

Podemos dividir a descrição do criptosistema de Paillier em 5 partes: geração de chaves, encriptação, decriptação, propriedades homomórficas e segurança.

2.1 Geração de Chaves

A geração de chaves neste criptosistema é se baseia na seleção aleatória de números primos grandes, ou seja, primos com 1024 bits ou mais. Neste projeto foram usados dois primos de 1536 bits cada. Como os primos possuem tamanhos iguais, podemos usar um algorítimo de geração de chaves mais simples[1], mas cuja segurança permanece inalterada. O processo de geração pode ser dividido em quatro etapas:

1. Escolha dois primos p e q de forma aletória e independente.
2. Compute $n = pq$ e $\lambda = \phi(n)$, onde $\phi(n) = (p-1)(q-1)$ é a função totiente de Euler.
3. Compute $g = n + 1$.
4. Compute $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$, onde $L(x) = \frac{x-1}{n}$. Note que $L(x)$ é uma função definida em $S = x < n^2 | x = 1 \bmod n$ e a operação de divisão não é modular.

Ao final do 4^o passo, a chave pública é definida por (n, g) e a chave privada por (λ, μ)

2.2 Encriptação

A encriptação no criptosistema Paillier é onde entra o aspecto probabilístico citado anteriormente. Nessa operação é selecionado um número aleatório r, que será usado no computo da cifra. Formalmente, seja $m \in \mathbb{Z}_n$ a mensagem a ser encriptada e $r \in \mathbb{Z}_n^*$ um número aleatoriamente escolhido. A cifra $c \in \mathbb{Z}_{n^2}^*$ é calculada por:

$$c = g^m r^n \bmod n^2$$

[1] Veja apêndice A para o algorítimo do caso geral

2.3 Decriptação

A decriptação neste caso pode ser descrita como uma simples exponeciação módulo n. De modo formal, seja $c \in \mathbb{Z}^*_{n^2}$ a cifra. Temos que a mensagem $m \in \mathbb{Z}_n$ é obtida por:

$$m = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$$

2.4 Propriedades Homomorficas

Homomorfismo em álgebra é definido como um mapeamento que preserva a estrutura entre duas estruturas algébricas. No caso do criptosistema de Paillier, encontramos duas propriedades homomórficas: adição e multiplicação.

2.4.1 Homomorfismo na Adição de Mensagens

Seja $E(m_i)$ a operação de encriptação descrita acima e $D(c_i \bmod n^2)$ a operação de decriptação descrita acima, onde $c_i = E(m_i)$. A equivilência nos dá um homomorfismo na adição de mensagens:

$$D(E(m_1) \cdot E(m_2) \bmod n^2) = m_1 + m_2 \bmod n$$

Corolário: Seja $E(m)$ uma cifra com obtida a partir da operação de encriptação descrita acima. Temos que:

$$D(E(m)^k \bmod n^2) = k \cdot m \bmod n$$

2.4.2 Homomorfismo na Multiplicação de Mensagens

Usando o colorário acima, podemos definir o homomorfismo da multiplicação entre duas mensagens:

$$\left.\begin{array}{l} D(E(m_1)^{m_2} \bmod n^2) \\ D(E(m_2)^{m_1} \bmod n^2) \end{array}\right\} = m_1 m_2 \bmod n$$

2.5 Segurança do Criptosistema de Paillier

O criptosistema de Paillier é baseado na Suposição do Composto Residual Decisacional (DCRA, sigla em inglês), que de modo informal, diz que dado um inteiro composto n e um inteiro z, é difícil decidir se z é um n-resíduo módulo n^2 ou não. Em notação matemática,

$$z \equiv y^n \pmod{n^2}$$

Devido a essa suposição ser considerada computacionalmente intratável, o criptosistema de Paillier é dito semanticamente seguro, ou seja, conhecimento de uma cifra e do tamanho de uma mensagem desconhecida não dá nenhuma informação extra sobre a mensagem cifrada que seja facilmente extraída.

Devido às propriedades homomórficas apresentadas acima, o sistema é, no entanto, maleável, ou seja, é possível a um adversário transformar uma cifra em outra cifra que decripta uma mensagem relacionada. Em geral, essa propriedade não desejada em um sistema de criptografia, mas existem aplicações como sistemas de votação ou dinheiro virtual que se beneficiam de tal propriedade.

2.6 Aceleração da Decriptação pelo uso do Teorema Chinês do Resto

Uma melhoria que pode ser feita na decriptação é o uso do teorema chinês do resto (CRT). Com o uso do CRT, em vez de realizar um cálculo de módulo n, realizamos dois cálculos, um de módulo p e um de módulo q, onde p e q são os fatores primos de n. Para isso, definimos as seguintes funções:

$$L_p(x) = \frac{x-1}{p} \qquad L_q(x) = \frac{x-1}{q}$$

onde L_p é definida sobre $S_p = \{x < p^2 | x = 1 \bmod p\}$ e L_q sobre $S_q = \{x < q^2 | x = 1 \bmod q\}$. Usando essas definições, realizamos os seguintes cálculos:

$$m_p = L_p(c^{p-1} \bmod n^2) \cdot h_p \bmod p$$

$$m_q = L_q(c^{q-1} \bmod n^2) \cdot h_q \bmod q$$
$$m = \text{CRT}(m_p, m_q) \bmod pq$$

onde

$$h_p = L_p(g^{p-1} \bmod n^2)^{-1} \bmod p$$
$$h_q = L_q(g^{q-1} \bmod n^2)^{-1} \bmod q$$

3 Implementação

A implementação do criptosistema foi realizada na linguagem Python 3.4 devido à sua facilidade em lidar com números de grande ordem. Como dito anteriormente, o tamanho em bits padrão selecionado para os primos foi 1536 bits. Esse tamanho pode ser mudado na hora da geração da chave se desejado. Para a geração da chave foram implementados dois métodos, o método geral[2] e o método mais simples geração de chaves. Nessa seção falaremos somente do método simples, aqui chamado `generateKeysSimple`, que pode ser vista abaixo:

```
def generateKeysSimple(prime_lenght=1536):

    p = utils.getPrime(prime_lenght)
    q = utils.getPrime(prime_lenght)

    n = p * q
    n_lambda = (p-1) * (q-1)

    g = n + 1

    n_mu = utils.inverseMod(n_lambda, n)

    public_key = PaillierPublicKey(n,g)
    private_key = PaillierPrivateKey(n_lambda, n_mu, p, q, public_key)

    return public_key, private_key
```

Atente ao uso da biblioteca `utils`, onde foram definidas várias funções auxiliares, como GCD, LCM e CRT, além de potência e inverso modular. As chaves são armazenadas em objetos próprios, aqui chamados de `PaillierPublicKey` e `PaillierPrivateKey`. Abaixo podemos ver a implementação da chave pública e seus métodos:

```
class PaillierPublicKey:

    def __init__(self, n, g):
        self.n = n
        self.g = g
        self.n_squared = self.n**2

    def encrypt(self, m):
        r = utils.getRandomInZ_N(self.n)

        g_m = utils.powMod(self.g, m, self.n_squared)
        r_n = utils.powMod(r, self.n, self.n_squared)

        cipher = (g_m * r_n) % self.n_squared

        return int(cipher)
```

[2]Ver apêndice A para detalhes e implementação

Note que a função de encriptação pertence à classe da chave pública. Do mesmo modo, a função de decriptação pertence à classe da chave privada, que é composta pelos elementos da chave privada e pela chave pública. Abaixo está a implementação de decriptação presente na classe `PaillierPrivateKey`:

```
def decrypt(self, c):
    c_lambda = utils.powMod(c, self.Lambda, self.n_squared)
    l = utils.Lfunction(c_lambda, self.public_key.n)

    message = (l * self.mu) % self.public_key.n

    return int(message)
```

Uma tentativa de se implementar a decriptação usando o CRT foi feita, porém sem sucesso. Abaixo se encontra o código da implementação do método usando CRT e os resultados encontrados no uso deste método podem ser vistos na próxima seção.

```
def decryptCRT(self, c):
    c_lambda1 = utils.powMod(c, self.p - 1, self.p ** 2)
    m_p = (utils.Lfunction(c_lambda1, self.p) * self.h_p) % self.p

    c_lambda2 = utils.powMod(c, self.q - 1, self.q ** 2)
    m_q = (utils.Lfunction(c_lambda2, self.q) * self.h_q) % self.q

    message = utils.crt([m_p],[m_q]) % (self.public_key.n)

    return int(message)
```

Outros detalhes da implementação relevates é o modo como as chaves são armazenadas no programa de demonstração e como a cifra é tratada. As chaves, após geradas, são armazenadas em JSON. A chave pública tem, por exemplo, a seguinte estrutura:

```
{
  ‘‘g’’: (inteiro),
  ‘‘n’’: (inteiro)
}
```

A chave privada é armazenada de forma similar, porém com campos extras para a chave pública e para os primos p e q. Por fim, abordamos como a mensagem é tratada durante os processos de encriptação e decriptação. Inicialmente a mensagem é lida como uma string de bytes, para então ser convertida para uma string em hexadecimal e por fim para uma um inteiro. Desse modo, podemos cifrar mensagens de até cerca de 50 caractéres, caso contrário o limite de n^2 será ultrapassado. O caminho inverso é realizado na hora de se obter a mensagem a partir de uma cifra.

Vale dizer que inicialmente estava-se realizando a encriptação caractér a caractér, o que nos levava a um tempo de execução extremamente alto para mensagens com mais de 100 caractéres. Abaixo mostramos o resultado da nova implementação.

Por fim, um segundo arquivo de demo foi realizado para o teste da propriedade homomórfica de adição do criptosistema. Abaixo pode-se ver as operações realizadas:

```
def main():
    public, private = paillier.generateKeysSimple()

    a = input("Enter the first integer: ")
    cipher_a = public.encrypt(int(a, 0))

    b = input("Enter the second integer: ")
    cipher_b = public.encrypt(int(b, 0))
```

```
    summation = private.decrypt(cipher_a * cipher_b)

    print("The summation of 'a' and  'b' can be obtained by
          the decryptation of enc(a)*enc(b): " + str(summation))
```

4 Resultados

Os testes foram executados um laptop com processador Intel Core i7 4700Q e 8GB de RAM. Para a medição do tempo de execução foi usado o comando `time` do GNU/Linux.

O primeiro teste foi a geração de um par de chaves usando o comando `python3 demo.py -generate`[3]. Abaixo pode-se ver o tempo corrido na geração:

```
fesoliveira@deadelus$ time python3 demo.py --generate

real    0m0.472s
user    0m0.463s
sys 0m0.012s
```

Em seguida, foi realizada a encriptação da seguinte cadeia de caractéres:

```
Isso é uma mensagem em texto pleno.
```

O tempo corrido da encriptação desta mensagem pode ser visto abaixo:

```
fesoliveira@deadelus$ time python3 demo.py --encrypt public_key.json plaintext.txt

real    0m0.160s
user    0m0.151s
sys 0m0.012s
```

Por fim, foi feita a decriptação da cifra gerada pela operação acima, e seu tempo pode ser visto abaixo:

```
fesoliveira@deadelus$ time python3 demo.py --decrypt private_key.json ciphertext.txt

real    0m0.081s
user    0m0.073s
sys 0m0.008s
```

A decriptação se mostrou bem sucedida, e o diff entre o arquivo de saída e o arquivo de entrada pode ser visto abaixo:

```
fesoliveira@deadelus$ diff -s output/plaintext.txt input/mensagem_original.txt
Files output/plaintext.txt and input/mensagem_original.txt are identical
```

Na execução da decriptação usando o CRT, no entanto, um resultado estranho ocorreu. Nos passos onde são calculados m_p e m_q, ao invés obtermos inteiros aparentemente aleatórios e igualmente grandes, foi-se obtido a mensagem original como resultado de ambos os cálculos. O motivo pelo qual isso ocorreu não foi identificado e mais testes não puderam ser executados devido a isso.

Usando o segundo arquivo de teste pudemos realizar o teste da propriedade homomórfica de adição do criptosistema de Paillier. Nele, encriptamos dois inteiros, a e b, e realizamos a multiplicação das cifras correspondentes seguido da decriptação deste. O resultado em todos os testes mostrou confirmação da propriedade homomórfica de adição.

[3] Ver apêndice A para resultados da geração de caso geral

5 Conclusão

Com esse projeto pudemos realizar a implementação de um sistema de criptografia assimétrico e ver seu funcionamento em primeira mão, assim como fazer o estudo de suas propriedades e nuâncias. Pudemos também realizar a confirmação do homomorfismo da adição de mensagens do criptosistema, assim como ver o tempo de execução de suas operações. No entanto, nos deparamos com um problema na implementação da decriptação usando o CRT, onde encontramos uma anomalia no cálculo de m_p e m_q, que nos retornavam já a mensagem decriptada. O motivo de isso acontecer, apesar de ter sido investigado a fundo, não foi encontrado.

Apêndice A - Processo Geral de Geração de Chaves

Nesse apêndice falaremos sobre o caso geral de geração de chaves no criptosistema de Paillier.

Etapas da Geração de Chaves

O caso geral da geração de chaves é usado quando p e q são suficientemente grandes e espaçados, de modo que a diferença entre seus tamanhos é significativa. Neste caso, um processo diferente do visto na seção 2 deve ser usado:

1. Escolha dois primos p e q de forma aletória e independente.
2. Compute $n = pq$ e $\lambda = lcm(p-1, q-1)$, onde $lcm(x, y)$ é a função que retorna o menor múltiplo comum entre x e y.
3. Escolha aleatoriamente um inteiro g, onde $g \in \mathbb{Z}^*_{n^2}$.
4. Compute $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$, onde $L(x) = \frac{x-1}{n}$. Note que $L(x)$ é uma função definida em $S = x < n^2 | x = 1 \bmod n$ e a operação de divisão não é modular.

Implementação do Caso Geral

```
def generateKeys(prime_lenght=1536):
    p = utils.getPrime(prime_lenght)
    q = utils.getPrime(prime_lenght)

    n = p * q
    n_squared = n ** 2

    n_lambda = utils.computeLCM(p,q)

    g = utils.getRandomInZ_N2(n)
    n_mu = utils.inverseMod(utils.Lfunction(utils.powMod(g, n_lambda, n_squared), n), n)

    while n_mu == None:
        g = utils.getRandomInZ_N2(n)
        n_mu = utils.inverseMod(utils.Lfunction(utils.powMod(g, n_lambda, n_squared), n), n)

    public_key = PaillierPublicKey(n,g)
    private_key = PaillierPrivateKey(n_lambda, n_mu, p, q, public_key)

    return public_key, private_key
```

Observe que nesse caso devemos escolher outro g caso não sejamos capazes de calcular μ devido à inexistência do inverso modular.

Resultados

A seguir temos o tempo corrido na geração das chaves usando o caso geral (ambas as chaves com tamanhos iguais para fim de comparação):

```
fesoliveira@deadelus$ time python3 demo.py --generate -c

real    0m0.526s
user    0m0.512s
sys 0m0.012s
```

Podemos ver que o tempo foi ligeiramente mais longo comparado com a geração simples devido à necessidade de se gerar um terceiro número aleatório e possivelmente à possivel repetição dessa escolha caso o inverso modular não exista para o valor inicialmente escolhido para g.