Project - Private Information Retrieval

Thomas RENAUD et Louis RICHARD - OCC3

25/03/2025

Applied cryptography

Sommaire

Report : Vous pouvez retrouver tous nos codes sources sur notre projet GitHub - PIR.
lien : *https://github.com/T1CKG0LD/PIR*

1 Paillier Cryptosystem

L'objectif est de permettre à un client de récupérer un élément d'une base de données sans révéler son choix au serveur.
Principe du protocole PIR : le client génère une clé *paillier* et chiffre un vecteur contenant des 0 sauf à l'index souhaité (chiffré en 1). Puis, le serveur effectue un produit scalaire homomorphe entre ce vecteur et la base de données et envoie le résultat chiffré au client. Le client peut alors déchiffrer le résultat pour récupérer l'élément souhaité.

1.1 Implementation

On complète alors le fichier paillier.py, qui nécessite le fichier utils.py :

```
from utils import *
from Crypto.Util import number
import gmpy2
```

Comme demandé, nous importons la bibliotheque gmpy2, qui permet de manipuler efficacement des très grands nombres et d'effectuer des opérations cryptographiques complexes de manière rapide et précise.

```
class Paillier:
    def __init__(self, bits):
        self.bits = bits
        self.keyGen(bits)

    def keyGen(self, bits):

        while True:
            p = number.getPrime(bits)
            q = number.getPrime(bits)
            n = p * q
            if gmpy2.gcd(n, (p-1)*(q-1)) == 1:
                break
```

1.1.1 Génération des clés

On génère deux nombres premiers p et q of nb bits bits each and we ensure that they have the same bit length. De plus, on vérifie que $gcd(n, \varphi(n)) = 1$ où $n = p \cdot q$ et $\varphi(n) = (p-1)(q-1)$.

```
        g = n + 1
        lambda_val = (p-1) * (q-1)
        mu = gmpy2.invert(lambda_val, n)

        self.pk = (n, g)
        self.sk = (lambda_val, mu)
```

On calcule $g = n + 1$, $\lambda = \varphi(n)$, et $\mu = \lambda^{-1} \pmod{n}$ qui est l'inverse modulaire de λ modulo n. On stocke ensuite ces valeurs dans deux variables *self.pk* et *self.sk*.

```
    def encrypt(self, message: int):
        n, g = self.pk
```

```
        if message < 0 or message >= n:
            raise ValueError("Error out of range")
```

On rajoute une boucle *if* qui fait attention à ce que la valeur soit bien dans la plage $[0, n-1]$. On s'interesse alors à la fonction de chiffrement.

1.1.2 Encryption

```
        while True:
            r = number.getRandomRange(1, n)
            if gmpy2.gcd(r, n) == 1:
                break
```

On génère un nombre aléatoire r tel que $0 < r < n$ et $gcd(r,n) = 1$.

```
        g_m = gmpy2.powmod(g, message, n**2)
        r_n = gmpy2.powmod(r, n, n**2)
        c = gmpy2.mul(g_m, r_n) % (n**2)

        return c
```

On calcule le ciphertext $c = g^m \cdot r^n \pmod{n^2}$, grâce à la fonction **powmod** de gmpy et **mul** qui multiplie g^m et r^n, puis applique un modulo n^2.

1.1.3 Decryption

```
    def decrypt(self, ciphertext: int):
        n, g = self.pk
        lambda_val, mu = self.sk

        def L(x):
            return gmpy2.f_div(x-1, n)

        c_lambda = gmpy2.powmod(c, lambda_val, n**2)
        m = gmpy2.mul(L(c_lambda), mu) % n

        return int(m)
```

On définit la fonction $L(x) = (x-1)/n$, le quotient de la division euclidienne grâce à la fonction **f_div()**, puis on calcule $m = L(c^\lambda \pmod{n^2}) \cdot \mu \pmod{n}$.

1.1.4 Test

On peut alors tester notre implémentation et vérifier qu'elle fonctionne correctement.

```
phe = Paillier(1024)

m = "Trying to encrypt this message using Paillier"
c = phe.encrypt(string_to_int(m))
assert(int_to_string(phe.decrypt(c)) == m)
print("Test réussi: le message déchiffré correspond au message original.")
```

1.2 Questions

Question 2 : Given ciphertexts c_1 and c_2 of messages m_1 and m_2, what can you say about $c_1 \cdot c_2$ mod n2 ? Test your answer by adding an adequate test case in the file paillier.py.

Le produit de deux cyphertext est un cyphertext de la somme des messages :

$$c_1 \cdot c_2 \pmod{n^2} = Enc(m_1 + m_2) \tag{1}$$

On l'observe facilement (cf. addition_homomorphic_test.py) :

```
def test_homomorphic_addition(phe):
    m1 = 42
    m2 = 123
    c1 = phe.encrypt(m1)
    c2 = phe.encrypt(m2)

    n = phe.pk[0]
    product = (c1 * c2) % (n**2)

    decrypted = phe.decrypt(product)

    assert decrypted == (m1 + m2) % n
    print("Test d'addition homomorphique réussi")

test_homomorphic_addition(phe)
```

```
invite/>python addition_homomorphic_test.py
Test d'addition homomorphique réussi
invite/>
```

Question 3 : Given a ciphertext c_1 of a message m_1 and a plaintext message m_2, what can you say about $c_1 \cdot g^{m_2} \bmod n^2$? Test your answer by adding an adequate test case in the file paillier.py.

Multiplier un cyphertext par $g^{m_2} \bmod n^2$ équivaut à ajouter m_2 au message :

$$c_1 \cdot g^{m_2} \pmod{n^2} = Enc(m_1 + m_2) \tag{2}$$

```
def addition_constant_test(self):
    m1 = 42
    m2 = 123
    c1 = self.encrypt(m1)
    n, g = self.pk

    #multiplication par g^m2
    g_m2 = gmpy2.powmod(g, m2, n**2)
    new_cipher = (c1 * g_m2) % (n**2)

    decrypted = self.decrypt(new_cipher)

    assert decrypted == (m1 + m2) % n
    print("Test d'addition avec constante réussi")

phe.addition_constant_test()
```

```
invite/>python addition_constant_test.py
Test d'addition avec constante réussi
invite/>
```

Vous pouvez retrouver ce test ici : addition_constant_test.py

Question 4 : Given a ciphertext c_1 of a message m_1 and a plaintext message m_2, what can you say about $c_1 m_2 \bmod n^2$? Test your answer by adding an adequate test case in the file paillier.py.

Élever un cyphertext à la puissance $m_2 \bmod n^2$ équivaut à multiplier le message par m_2 :

$$c_1^{m_2} \pmod{n^2} = Enc(m_1 \cdot m_2) \tag{3}$$

```python
def multiply_constant_test(self):
    m1 = 42
    m2 = 3
    c1 = self.encrypt(m1)
    n = self.pk[0]

    powered_cipher = gmpy2.powmod(c1, m2, n**2)

    decrypted = self.decrypt(powered_cipher)

    assert decrypted == (m1 * m2) % n
    print("Test de multiplication par constante réussi")

phe.multiply_constant_test()
```

```
invite/>python multiply_constant_test
Test de multiplication par constante réussi
invite/>
```

Vous pouvez retrouver ce test ici : multiply_constant_test.py

2 PIR Protocol

Le protocole PIR (Private Information Retrieval) permet à un client de récupérer un élément spécifique dans une base de données serveur sans révéler quel élément il souhaite consulter. Poru cela nous utilisons **Paillier** pour les propriétés homomorphiques (addition/multiplication) ainsi que trois composants principaux :

- *client.py*, qui énère les requêtes et déchiffre les réponses
- *server.py*, qui stocke la DB et traite les requêtes
- *exchanges.py*, qui orchestre les échanges

2.1 Questions

Question 1 : How to homomorphically compute the value of t ? Describe the necessary operations.

On traite ici du **calcul homomorphique de** t.
Le serveur calcule la valeur t permettant de récupérer l'élément $T[i]$ de manière sécurisée grâce aux propriétés du cryptosystème de Paillier.

Opérations Requises

1. **Produit Scalaire Homomorphique** :
   Le serveur reçoit le vecteur chiffré $v = (c_0, c_1, \ldots, c_{n-1})$ où :

$$c_j = \begin{cases} \text{Enc}(1) & \text{si } j = i, \\ \text{Enc}(0) & \text{sinon.} \end{cases}$$

   puis le serveur calcule le produit scalaire entre ce vecteur et la base de données T, **sans déchiffrer** :

$$t = \prod_{j=0}^{n-1} c_j^{T[j]} \mod n^2$$

   Cette opération utilise les propriétés homomorphiques de Paillier.

2. **Simplification** : Grâce aux propriétés de Paillier, on peut simplifier le produit ainsi,

$$t = \mathsf{Enc}\left(\sum_{j=0}^{n-1} \delta_{ij} T[j]\right) = \mathsf{Enc}(T[i]), \quad \text{où } \delta_{ij} = \begin{cases} 1 & \text{si } j = i, \\ 0 & \text{sinon.} \end{cases}$$

3. **Renvoi au Client** :
   Le serveur envoie t au client qui peut alors le déchiffrer pour obtenir $T[i]$.

A ENLEVER DANS LE RAPPORT, JUSTE POUR NOUS AIDER À COMPRENDRE :

2.1.1 Exemple

- **Base de données** : $T = [10, 20, 30, 40]$ $(n = 4)$
- **Index demandé** : $i = 2$ $(T[2] = 30)$
- **Requête client** : $v = [\mathsf{Enc}(0), \mathsf{Enc}(0), \mathsf{Enc}(1), \mathsf{Enc}(0)]$
- **Calcul du serveur** :

$$t = \mathsf{Enc}(0)^{10} \cdot \mathsf{Enc}(0)^{20} \cdot \mathsf{Enc}(1)^{30} \cdot \mathsf{Enc}(0)^{40} = \mathsf{Enc}(30)$$

Pourquoi cela fonctionne-t-il ?

Le protocole repose sur deux propriétés clés du cryptosystème de Paillier :

1. **Homomorphisme multiplicatif** :

$$\mathsf{Enc}(a)^b \equiv \mathsf{Enc}(a \cdot b) \mod n^2$$

   - Permet de multiplier une valeur chiffrée par une constante (ici $T[j]$).

2. **Homomorphisme additif** :

$$\mathsf{Enc}(a) \cdot \mathsf{Enc}(b) \equiv \mathsf{Enc}(a + b) \mod n^2$$

   - Permet d'accumuler les résultats via une multiplication des termes.

Isolation de l'élément recherché
Soit $v = [\mathsf{Enc}(0), \ldots, \mathsf{Enc}(1)$(en position $i, \ldots, \mathsf{Enc}(0)]$. Le calcul :

$$t = \prod_{j=0}^{n-1} \underbrace{c_j^{T[j]}}_{\mathsf{Enc}(\delta_{ij} \cdot T[j])} \equiv \mathsf{Enc}\left(\sum_{j=0}^{n-1} \delta_{ij} T[j]\right) \equiv \mathsf{Enc}(T[i])$$

où δ_{ij} est le symbole de Kronecker (1 si $j = i$, sinon 0).
Car seul $c_i = Enc(1)$ contribue au résultat final via $Enc(1)^{T[i]} = Enc(T[i])$. Les autres termes $Enc(0)^{T[i]} = Enc(0)$ n'ont pas d'incidence.

Confidentialité garantie
Le serveur ne voit que des valeurs chiffrées indistinguables (semantic security). La position i est masquée par la construction de v.

Question 2 : Which value does the client retrieve after decrypting the answer from the server ?

Lorsque le client déchiffre la réponse t envoyée par le serveur, il obtient **exactement l'élément** $T[i]$ de la base de données à l'index i qu'il souhait récupérer (et rien d'autre), sans que le serveur ne connaisse cet élément i.

Le protocole est détaillé question 1. À la suite de cela, le client applique sa clé privée pour obtenir $T[i]$:

$$\mathsf{Dec}(t) = T[i]$$

2.2 Implementation du Client PIR

Dans cette partie we create the file, disponible ici client.py.

2.2.1 Constructor

La classe **Client** permet de récupérer un élément d'une base de données serveur sans révéler l'index demandé.

```
from paillier import Paillier
from utils import *

class Client:
    def __init__(self, bits=1024):
        self.phe = Paillier(bits)
        self.pk = self.phe.pk
        self.sk = self.phe.sk
```

On initialise le client avec un cryptosystème Paillier, et on génère nos clés Paillier : une publique (n,g) qui nous sert à chiffrer et une privée (λ, μ) pour le déchiffrement.

2.2.2 Request method

On génère ensuite une requête PIR pour récupérer l'élément à l'index spécifié. On retourne la liste des $[Enc(0), ..., Enc(1), ..., Enc(0)]$ où seul l'élément à 'index' est chiffrer Enc(1), 0 pour tous les autres.
On fait également attention au cas où l'index ne serait pas inclus dans notre base de donnée.

```
    def request(self, db_size, index):
        if index < 0 or index >= db_size:
            raise ValueError(f"Index {index} hors limites (taille DB: {db_size})")

        request_vector = []
        for j in range(db_size):
            message = 1 if j == index else 0
            encrypted_bit = self.phe.encrypt(message)
            request_vector.append(encrypted_bit)

        return request_vector
```

2.2.3 DecryptAnswer method

On déchiffre ensuite la réponse du serveur pour obtenir l'élément demandé avec la clé secrète du client. Le serveur ayant fait un calcul homomorphique, le résultat est $\mathsf{Enc}(T[i])$.

```
    def decrypt_answer(self, encrypted_answer):
        decrypted_value = self.phe.decrypt(encrypted_answer)

        print(f"Réponse serveur déchiffrée: {decrypted_value}")
        return decrypted_value
```

2.3 Implémentation du Serveur PIR

On implémente server.py, le serveur PIR qui permet à un client de récupérer un élément précis dans la base de données sans que le serveur ne sache quel élément a été demandé.

2.3.1 Constructeur

Le constructeur génère une base de données aléatoire avec valeurs entre 1 et 2^{16} (classe Server, qui vérifie aussi que la taille de la database ne soit pas négative ce qui est impossible), que l'on stocke dans la variable **self.db** (*où _ représente l'index*).

```
import random
from utils import *

class Server:
    def __init__(self, db_size):
        if db_size <= 0:
            raise ValueError("DB size must be > 0")
        self.db = [random.randint(1, 2**16) for _ in range(db_size)]
```

2.3.2 Answer_request method

La méthode *answer_request* traite une requête PIR et retourne le résultat chiffré. À la suite du calcul homomorphique seul l'élément demandé ets inclus dans le résultat final (cf Paillier). On stocke la clé publique dans **client_pk**. Encore une fois, on veille à faire attention à la taille de database.
encrypted_answer = 1 est l'élément neutre pour la multiplication modulaire ; puis on réalise le calcul homomorphique du produit scalaire comme en partie 1.
La méthode retourne le résultat chiffré Enc$(DB[i])$ où i est l'index secret du client.

```
    def answer_request(self, request_vector, client_pk):
        n, g = client_pk
        if len(request_vector) != len(self.db):
            raise ValueError("Request size doesn't match DB size")

        encrypted_answer = 1
        for db_val, encrypted_bit in zip(self.db, request_vector):
            term = pow(encrypted_bit, db_val, n**2)
            encrypted_answer = (encrypted_answer * term) % (n**2)

        return encrypted_answer
```