UNIVERSITÉ
DE GENÈVE

University of Geneva
Department of Science
Master's Degree in Computer Science

Printemps 2025

Sécurité Avancée

Chiffrement homomorphe

Enseignant :
Eduardo Solana

Etudiant :
David Gabriel PAPA

5 mai 2025

Table des matières

1 Introduction

À l'heure où les données numériques sont présentes dans tous les domaines du quotidien, garantir leur sécurité sans compromettre leur utilité opérationnelle représente un défi majeur. Les méthodes de chiffrement classiques, bien qu'efficaces pour protéger l'intégrité et la confidentialité de ces données nécessitent généralement un déchiffrement préalable pour permettre tout traitement ou analyse. Cette étape intermédiaire crée une faille de sécurité potentielle, exposant temporairement les données sensibles à d'éventuelles intrusions.

C'est dans ce contexte que l'encryption homomorphique se distingue. Cette technique cryptographique permet d'effectuer des calculs directement sur des données chiffrées, sans jamais devoir les décrypter. Le résultat de ces opérations, une fois déchiffré, correspond exactement à ce que l'on aurait obtenu en travaillant sur les données en clair. Cela ouvre des perspectives radicalement nouvelles en matière de traitement sécurisé, notamment dans des domaines sensibles tels que la santé, la finance, l'intelligence artificielle ou l'administration publique.

Le chiffrement homomorphique a connu un tournant décisif en 2009 avec les travaux de Craig Gentry, qui proposa le premier schéma pleinement homomorphe. Depuis, les recherches se sont multipliées pour en améliorer l'efficacité et la faisabilité à grande échelle.

Dans ce projet, nous travaillerons d'abord sur des algorithmes homomorphique simple permettant d'effectuer généralement une seule opération mathématique par algorithme avant de manipuler un chiffrement plus complexe nous permettant l'utilisation de différentes opérations. Pour nous aider dans cette étape, nous utiliserons la librairie `TenSEAL` qui permet d'effectuer des opérations de chiffrement homomorphe, basée sur Microsoft SEAL. Elle offre une grande simplicité d'utilisation grâce à une API Python, tout en préservant l'efficacité grâce à l'implémentation de la plupart de ses opérations en C++.

Après avoir utilisé ces différents algorithmes, nous pourrons comparer leurs performances en terme de rapidité opérationnelle ainsi qu'en utilisation de mémoire.

2 Méthodes

Dans cette partie, nous verrons quelques fonctions utiles pour nos différentes clés et nous expliquerons les cinq algorithmes homomorphes utilisés avec leurs spécificités ainsi que le fonctionnement de la librairie `TenSEAL`. Toute l'implémentation des différentes méthodes se fera sur un notebook que vous pouvez retrouver en entier sur ce dépot Git : DavidGabrielPapa/securité_avancée

2.1 Fonctions utiles

Nos différentes méthodes d'encryptions possèdent souvent des opérations similaire entre elles qui seront décrites dans cette partie.

2.1.1 Algorithme d'Euclide étendu

L'algorithme d'Euclide étendu est une version améliorée de l'algorithme d'Euclide classique, utilisé pour calculer le plus grand commun diviseur et qui permet en plus de trouver les coefficients de Bézout : deux entiers x et y tels que :

$$pgcd(a, b) = xa + by$$

L'algorithme d'Euclide étendu est essentiel en cryptographie notamment dans notre cas où il nous servira à calculer des inverses modulo. Si a et b sont premiers entre eux ($pgcd(a, b) = 1$), alors x est l'inverse de a modulo b :

$$ax \equiv 1 \mod b$$

Nous avons donc implémenté cette fonction qui nous retourne le $pgcd(a, b)$, la valeur x et la valeur y :

```
def algo_Euclide(a, b) :
    ri_moins_1 = a
    ri = b
    si_moins_1 = 1
    si = 0
    ti_moins_1 = 0
    ti = 1
    qi = a // b

    ri_plus_1 = 1

    while (ri_plus_1 != 0) :
        ri_plus_1 = ri_moins_1 - (qi * ri)
        si_plus_1 = si_moins_1 - (qi * si)
        ti_plus_1 = ti_moins_1 - (qi * ti)
        if (ri_plus_1 != 0) :
            qi = ri // ri_plus_1
```

```
            ri_moins_1 = ri
            ri = ri_plus_1

            si_moins_1 = si
            si = si_plus_1

            ti_moins_1 = ti
            ti = ti_plus_1

    return ri, si, ti
```

2.1.2 Exponentiation rapide

L'exponentiation rapide permet de calculer une puissance a^b beaucoup plus vite que si on multipliait a par lui-même b fois. Dans un contexte cryptographique, on veut souvent calculer de grands nombres :

$$a^b \mod n$$

L'idée est de réduire le nombre d'opérations en se reposant sur la décomposition binaire de l'exposant b. Pour calculer a^{13}, on peut décomposer notre opération comme ceci :

$$13 = 1101_2 = 8 + 4 + 0 + 1 \Rightarrow a^{13} = a^8 \cdot a^4 \cdot a^1$$

Donc au lieu de faire 12 multiplications $(a \cdot a \cdot ... \cdot a)$, on fait seulement $\log_2(b)$ étapes grâce aux carrés successifs. De plus, lorque que l'on applique un modulo à une multiplication, cela revient à calculer la multiplication de leur modulo :

$$a^{13} \mod n = ((a^8 \mod n) \cdot (a^4 \mod n) \cdot (a^1 \mod n)) \mod n$$

On peut ainsi calculer les carrés successifs de a modulo n à l'avance avant de les multiplier pour obtenir un résultat rapide avec des valeurs extrêmement grandes. On a donc une fonction qui prend en argument les valeurs a, b et le modulo n :

```
def expo_rapide(n, puissance, mod) :
    nombre = bin(puissance)[2:] # on transforme la puissance en binaire pour savoir
    ↪  jusqu'a quelle puissance on doit faire l'expo rapide
    nb = len(nombre)
    liste_puiss = [0 for x in range(nb)]
    liste_puiss[0] = n
    for b in range(1, nb) : # on calcule la valeur pour chaque puissance de 2
        liste_puiss[b] = pow(liste_puiss[b - 1], 2) % mod
    total = 1
    for p in range(0, nb) : # on multiplie chaque valeur correpondante a notre tableau
    ↪  en appliquant le modulo a chaque fois
        if (nombre[p] == '1') :
```

```
            total = (total * liste_puiss[nb - p - 1]) % mod
    return total
```

2.1.3 Test de primalité de Miller-Rabin

Le test de primalité de Miller-Rabin est un algorithme probabiliste qui permet de vérifier rapidement si un nombre est probablement premier. Il est souvent utilisé en cryptographie, notamment dans notre cas pour vérifier la primalité de grands nombres. Etant donné qu'il s'agit d'un test probabiliste, il nous dira si un nombre n est composé ou s'il est probablement premier. On peut réduire le risque d'erreur en répetant plusieurs fois notre opération.

Pour trouver si un nombre n est premier, on part du petit théorème de Fermat :

Si n est un nombre premier et a un entier tel que $1 < a < n - 1$, alors :

$$a^{n-1} \equiv 1 \mod n$$

Nous allons donc générer un a aléatoire entre $1 < a < n-1$ avant de vérifier si l'équation précédente est satisfaite. Si elle ne l'est pas, on peut affirmer que n n'est pas premier, dans le cas contraire, on va répéter l'opération plusieurs fois afin de réduire le risque qu'une combinaison de a et n nous donne un résultat corrompu par coïncidence. Plus le nombre de répétition est grand, plus le test sera fiable cependant il prendra aussi plus de temps :

```
def is_prime(n, k=20) :
    if n < 2 :
        return False
    for _ in range(k) : # nombre d'itérations
        a = random.randint(2, n)
        if expo_rapide(a, n - 1, n) != 1 :
            return False
    return True
```

2.1.4 Décomposition en produit de facteurs premiers

La décomposition en produit de facteurs premiers est une opération mathématique qui consiste à écrire un entier naturel n comme produit de nombres premiers. Etant donné que les nombres premiers peuvent apparaître plusieurs fois dans une décomposition, on utilisera un set pour les lister étant donné que seul les facteurs nous intéressent et pas leur nombre d'apparition. On aura par exemple :

$$72 = 8 \cdot 9 = 2^3 \cdot 3^2 \Rightarrow prime\_factors(72) = \{2, 3\}$$

On écrira notre fonction comme ceci :

```python
def prime_factors(n) :
    factors = set()
    d = 2
    while d * d <= n : # si n a un facteur, il est <= sqrt(n)
        while n % d == 0 :
            factors.add(d)
            n //= d
        d += 1
    if n > 1 :
        factors.add(n)
    return factors
```

2.2 Chiffrement RSA

Le chiffrement RSA (du nom de ses inventeurs : Rivest, Shamir et Adleman) est l'un des algorithmes de cryptographie asymétrique les plus connus et les plus largement utilisés dans le monde. Il repose sur des principes mathématiques solides, en particulier la difficulté de factoriser de grands nombres premiers, ce qui le rend particulièrement adapté pour sécuriser les communications sur des réseaux ouverts.

Contrairement à la cryptographie symétrique, où la même clé est utilisée pour chiffrer et déchiffrer les données, RSA utilise deux clés distinctes : une clé publique pour chiffrer les données, une clé privée pour les déchiffrer.

Ces deux clés sont mathématiquement liées, mais il est pratiquement impossible de déduire la clé privée à partir de la clé publique, tant que les nombres utilisés sont suffisamment grands.

2.2.1 Génération des clés

Afin d'obtenir nos clés, il faut suivre certaines étapes :

1. On choisit deux grands nombres premiers, notés p et q.

   ```
   p = random.getrandbits(bits)
   q = random.getrandbits(bits)
   ```

2. On calcule leur produit : $n = p \cdot q$. Ce nombre sera utilisé comme modulo pour les opérations de chiffrement et déchiffrement.
3. On calcule $\lambda(n)$ qui correspond à l'indicatrice de Carmichael et s'obtient en calculant le plus petit multiple commun entre $p-1$ et $q-1$ ($\lambda(n) \mod (p-1) \equiv \lambda(n) \mod (q-1) \equiv 0$).

   ```
   lambd = np.lcm(p-1, q-1) # Plus petit multiple commun
   ```

4. On choisit un entier e, premier avec $\phi(n) = (p-1) \cdot (q-1)$ et inférieur à celui-ci ($e < \phi(n)$).
5. On calcul d, l'inverse modulaire de e modulo $\phi(n)$, c'est-à-dire que $d \cdot e \equiv 1 \mod \phi(n)$. On peut facilement le calculer avec l'algorithme d'Euclide étendu.

```
r, s, t = algo_Euclide(lambd, e)
d = t % lambd
```

Ainsi, on obtient la clé publique qui est le couple (e, n) et la clé privée qui est le couple (d, n).

2.2.2 Chiffrement

Maintenant que nous possèdons notre clé public, nous pouvons chiffrer notre message m. L'unique prérequis est que m soit inférieur à n ($m < n$). Nous pouvons donc simplement générer notre message chiffré c comme ceci :

$$c = m^e \mod n$$

2.2.3 Déchiffrement

Afin de déchiffrer notre message, il nous suffit d'utiliser notre d :

$$m = c^d \mod n$$

Car le petit théorème de Fermat nous donne :

$$c^d \equiv m^{ed} \mod n$$

Avec $ed = 1 + k(p-1)(q-1)$ pour un entier k :

$$m^{ed} \equiv m^{1+k(p-1)(q-1)} \equiv m \cdot m^{k(p-1)(q-1)} \equiv m \mod p \equiv m \mod q$$

L'entier $m^{ed} - m$ est donc un multiple de p et de q, qui sont premiers distincts, donc de leur produit $pq = n$.

2.2.4 Spécificité homomorphique

Etant donné un message m, son chiffrement RSA sera $c = \mathcal{E}(m) = m^e \mod n$. Cette méthode de chiffrement étant assez simple, si l'on souhaite multiplier deux valeurs entre elles, il suffit de multiplier les deux nombres encryptés entre eux avant d'y appliquer le modulo n :

$$\mathcal{E}(m_1) \cdot \mathcal{E}(m_2) = m_1^e \cdot m_2^e \mod n = (m_1 m_2)^e \mod n = \mathcal{E}(m_1 \cdot m_2)$$

Dans notre code, nous avons donc une fonction qui permet d'effectuer cette opération :

```
def multiply_homomorphic(c1, c2, n) :
    return (c1*c2) % n
```

2.3 Cryptosystème de ElGamal

Le cryptosystème d'ElGamal est une méthode de chiffrement asymétrique inventée par Tahar ElGamal. Il repose sur la difficulté du problème du logarithme discret qui est que pour un certain nombre de groupes, on ne connait pas d'algorithme efficace pour le calcul du logarithme discret, alors que celui de la réciproque, l'exponentiation, se réalise en un nombre de multiplications logarithmique en la taille de l'argument.

Comme RSA, ElGamal utilise une paire de clés, une clé publique et une clé privée. Toutefois, il est probabiliste : cela signifie que le même message chiffré plusieurs fois donnera à chaque fois un résultat différent, ce qui renforce la confidentialité contre certaines attaques.

2.3.1 Génération des clés

Pour générer nos clés, on agit comme tel :

1. On choisit un grand nombre premier p, on défini q tel que $q = p - 1$ et g un générateur du groupe multiplicatif $\mathbb{Z}_p^*$. Pour calculer ce dernier, on utilisera la librairie `sympy`.

```
p = random.getrandbits(bits)
q = p - 1

g = primitive_root(p) # fonctiuon de sympy
```

2. On choisit une clé privée x au hasard entre 1 et $q-1$ ($1 \leq x \leq q-1$).
3. On calcule $h = g^x \mod p$.

```
h = expo_rapide(g, x, p) # exponentielle rapide pour les grands nombres
```

On obtient donc notre clé public (p, q, g, h) et notre clé privée x.

2.3.2 Chiffrement

Pour un message chiffré M, on doit d'abord l'adapter pour qu'il fasse partie du groupe cyclique $\mathbb{Z}_p^*$. Dans notre code, on appliquera simplement un modulo p pour obtenir notre message m, nous perdons des informations et après reconstruction du message nous aurons $M = m + k \cdot p$ avec $k \in \mathbb{N}$ et nous ne pourrons pas retrouver exactement notre M d'origine.

```
# Mapping M vers m  Z*_p
m = M % p
if m == 0 :
    raise ValueError("Le message ne peut pas être 0 dans Z*_p")
```

Après cette première étape, on peut générer un nombre y aléatoire compris entre 1 et $q-1$ ($1 \leq y \leq q-1$). A l'aide de ce nombre, on va pouvoir calculer deux textes encryptés :

$$c_1 = g^y \mod p$$

$$c_2 = m \cdot h^y \mod p$$

On obtient donc un tuple de message chiffré (c_1, c_2).

2.3.3 Déchiffrement

Pour retrouver m à partir de (c_1, c_2) :

On peut calculer s^{-1} grâce au théorème de Lagrange :

$$s^{-1} = c_1^{q-x} \mod p$$

Ce qui nous permet de récupérer le message m :

$$m = c_2 \cdot s^{-1} \mod p$$

Car :

$$c_2 \cdot s^{-1} = m \cdot h^y \cdot c_1^{q-x} = m \cdot g^{xy} \cdot g^{y(q-x)} = m \cdot g^{xy} \cdot g^{qy} \cdot g^{-xy} = m \cdot g^{qy} \mod p \equiv m \mod p$$

2.3.4 Spécificité homomorphique

Les propriétés homomorphe du chiffrement de ElGamal sont assez similaires à celles de RSA. Une des grosses différences réside dans le fait que le chiffrement de ElGamal pour un message m nous donne deux messages chiffrés : $c = \mathcal{E}(m) = (g^y, m \cdot h^y)$. En multipliant chacune des deux parties avec celle correspondante au deuxième message chiffré, on obtient le produit de nos deux messages d'origine :

$$\mathcal{E}(m_1) \cdot \mathcal{E}(m_2) = (g^{y_1}, m_1 \cdot h^{y_1}) \cdot (g^{y_2}, m_2 \cdot h^{y_2}) = (g^{y_1+y_2}, (m_1 m_2) \cdot h^{y_1+y_2}) = \mathcal{E}(m_1 \cdot m_2)$$

Comme pour RSA, nous avons implémenté une fonction pour effectuer cette opération :

```
def multiply_homomorphic(c1, c2) :
    c_11, c_12 = c1
    c_21, c_22 = c2
    return (c_11 * c_21, c_12 * c_22)
```

2.4 Cryptosystème de Goldwasser-Micali

Le cryptosystème de Goldwasser-Micali, proposé par Shafi Goldwasser et Silvio Micali, est l'un des premiers exemples d'un chiffrement probabiliste en cryptographie asymétrique. Il repose sur la difficulté du problème de la résiduosité quadratique et a introduit des concepts fondamentaux pour la cryptographie moderne, notamment la notion de sécurité sémantique.

Contrairement à RSA ou ElGamal qui chiffrent directement des messages entiers, Goldwasser-Micali chiffre bit par bit, ce qui a une influence sur l'efficacité, mais offre un haut niveau de sécurité.

2.4.1 Génération des clés

Pour avoir nos clés, on suit ces étapes :

1. On choisit deux grands nombres premiers p et q.
2. On calcule $N = p \cdot q$.
3. On cherche un x qui satisfait la fonction du symbole de Legendre tel que : $\frac{x}{p} = \frac{x}{q} = -1$. On peut facilement calculer cela car : $a^{\frac{p-1}{2}} \equiv \frac{a}{p} \mod p$. On a donc notre code :

```
def legendre_symbol(a, p):
    return expo_rapide(a, (p - 1) // 2, p) if a % p != 0 else 0

x = random.randrange(2, N)
while not (legendre_symbol(x, p) == p-1 and legendre_symbol(x, q) == q-1):
    x = random.randrange(2, N)
```

Nous obtenons la clé publique (x, N) et la clé privée (p, q).

2.4.2 Chiffrement

Avant de pouvoir chiffrer notre message m, il faut le transformer en série de bits. Dans notre implémentation, on représentera cette série de bits sous forme de liste et pour éviter des soucis lors d'opérations homomorphes, on va retourner une liste de taille identique à chaque fois :

```
# Transforme un int en liste de bits
def int_to_bits(n, bit_length=8) :
    return [int(b) for b in bin(n)[2:].zfill(bit_length)]
```

Ensuite, nous pourrons itérer sur notre liste de bit (où i correspond au i^{eme} bit) pour l'encrypter comme ceci :

1. On génère un y_i entre 2 et $N-1$ ($2 \leq y_i \leq N-1$).
2. On calcule le bit encrypté : $c_i = y_i^2 \cdot x^{m_i} \mod N$

```
ciphertext = []

for bit in mess :
    # Générer y tel que gcd(y, N) = 1
    y = random.randint(2, N - 1)

    c = (expo_rapide(y, 2, N) * expo_rapide(x, bit, N)) % N
    ciphertext.append(c)
```

On obtient donc un message crypté *ciphertext* aussi sous forme de liste.

2.4.3 Déchiffrement

Avec la clé privée, on teste si c_i est un résidu quadratique modulo N, si oui, le bit déchiffré est 0 sinon 1. Pour cela, on vérifie que le symbole de Legendre du bit avec p et q soit de 1 :

$$m_i = \begin{cases} 0 & \text{pour} \quad c_i^{\frac{p-1}{2}} \mod p = c_i^{\frac{q-1}{2}} \mod q = 1 \\ 1 & \text{sinon} \end{cases}$$

```
    mess = []
    for c in ciphertext :
        bit = -1
        if legendre_symbol(c, p) == 1 and legendre_symbol(c, q) == 1 :
            bit = 0
        else :
            bit = 1
        mess.append(bit)
```

Enfin il suffit de convertir la liste de bit en leur valeur pour retrouver le message m originel :

```
# Transforme une liste de bits en int
def bits_to_int(bits) :
    return int("".join(map(str, bits)), 2)

m = bits_to_int(mess)
```

2.4.4 Spécificité homomorphique

Contrairement aux spécificités précédentes, le cryptosystème de Goldwasser-Micali opère sur les bits, ce qui signifie que sa capacité homomorphe sera une opération binaire. Pour un bit b, on aura son chiffrement $\mathcal{E}(b) = y^2x^b \mod N$, donc en effectuant une multiplication des bits chiffrés, on obtiendra un XOR de ceux-ci (0 si identiques et 1 si différents) :

$$\mathcal{E}(b_1) \cdot \mathcal{E}(b_2) = y_1^2 x^{b_1} \cdot y_2^2 x^{b_2} \mod N = (y_1 y_2)^2 x^{b_1+b_2} \mod N = \mathcal{E}(b_1 \oplus b_2)$$

Etant donné qu'on a nos bits dans des listes, on va itérer sur ces listes pour effectuer notre opération. On notera qu'il est important que nos liste aient la même longueur pour faire correspondre les bons bits entre eux, étape qui a été faite dans la partie du chiffrement :

```
def add_bit_homomorphic(c1, c2, n) :
    res = []
    for b1, b2 in zip(c1,c2) :
        val = (b1 * b2) % n
        res.append(val)
    return res
```

2.5 Cryptosystème de Benaloh

Le cryptosystème de Benaloh, proposé par Josh Benaloh, est une amélioration directe du cryptosystème de Goldwasser-Micali. Il vise à résoudre l'un des gros inconvénients de Goldwasser-Micali : au lieu de ne chiffrer qu'un seul bit à la fois, Benaloh permet de chiffrer des blocs plus grands, tout en conservant les avantages de la sécurité probabiliste et du chiffrement homomorphique.

Le cryptosystème de Benaloh utilise une structure mathématique similaire à Goldwasser-Micali, mais il permet de chiffrer un message qui appartient à un ensemble plus grand que juste 0,1. On peut choisir une valeur r qui permet de chiffrer $\log_2(r)$ bits en une seule opération. La sécurité repose sur la difficulté de distinguer des puissances r-ièmes dans un grand groupe modulo n.

2.5.1 Génération des clés

Avant de décrire la marche à suivre pour générer nos clés, il faut instancier une valeur r plus grande que 2 qu'on utilisera pour toutes nos opérations. Selon la valeur du r, il est possible que celui-ci ne remplisse pas les conditions qui suivront et qu'il faille le changer. Une fois choisit, nous pouvons suivre ces étapes pour obtenir nos clés :

1. On choisit deux grands nombres premiers p et q tel que $(p-1) \mod r = 0$, $gcd(r, \frac{p-1}{r}) = 1$ et $gcd(r, (q-1)) = 1$.

```
p = random.getrandbits(bits)
q = random.getrandbits(bits)
while not (is_prime(p) and math.gcd(r, (p-1)/r) == 1) :
    p = random.getrandbits(bits)
while not (is_prime(q) and math.gcd(r, q - 1) == 1) :
    q = random.getrandbits(bits)
```

2. On calcule $n = p \cdot q$ et $\phi = (p-1) \cdot (q-1)$.
3. On choisit un $y \in \mathbb{Z}_n^*$ tel que $y^{\frac{\phi}{r}} \not\equiv 1 \mod n$. Pour trouver cela et accelérer les calculs, on peut factoriser $r = p_1 p_2 ... p_k$ puis calculer $y^{\frac{\phi}{p_i}} \not\equiv 1 \mod n$:

```
r_factors = prime_factors(r) # set de facteurs premiers de r
y = random.randrange(2, n)

while True :
    y = random.randrange(2, n)
    if math.gcd(y, n) != 1 :
        continue # Reset si pas inversible
    valid = True
    for pi in r_factors :
        if expo_rapide(y, phi // pi, n) == 1 :
            valid = False
            break # Arrête la boucle for
    if valid :
        break # Sort du while
```

4. On définit $x = y^{\frac{\phi}{r}} \mod n$.

Ce procéder nous permet d'obtenir notre clé publique (y, n) et notre clé privée (ϕ, x).

2.5.2 Chiffrement

Pour chiffrer notre message m, en plus de la clé publique, nous avons besoin du r défini pour nos opérations.

On défini premièrement un $u \in \mathbb{Z}_n^*$ avant de calculer notre message chiffré c :

$$c = y^m u^r \mod n$$

2.5.3 Déchiffrement

Pour retrouver notre message clair à partir du message chiffré il faut commencer par calculer :

$$a = c^{\frac{\phi}{r}} \mod n$$

Ensuite pour retrouver notre message m on cherche un m tel que $m = \log_x(a)$ ce qui revient a chercher :

$$x^m \equiv a \mod n$$

```
a = expo_rapide(c, phi // r, n)

for m in range(r) : # on teste tous les m
    if expo_rapide(x, m, n) == a :
        return m
```

2.5.4 Spécificité homomorphique

Avec le chiffrement de Benaloh, contrairement à RSA et ElGamal, pour un message m et son encryption $c = \mathcal{E}(m) = g^m u^r \mod n$, la multiplication de deux messages chiffrés donnera la somme des messages d'origines modulo r :

$$\mathcal{E}(m_1)\cdot\mathcal{E}(m_2) = (g^{m_1}u_1^r)\cdot(g^{m_2}u_2^r) \mod n = g^{m_1+m_2}(u_1u_2)^r \mod n = \mathcal{E}(m_1+m_2 \mod r)$$

Pour ne pas se tromper lors d'opérations, on definira une fonction d'addition homomorphique comme ceci :

```
def add_homomorphic(c1, c2) :
    return c1 * c2
```

2.6 Cryptosystème de Paillier

Le cryptosystème de Paillier a été inventé par Pascal Paillier en 1999. Il est célèbre parce qu'il offre un chiffrement homomorphe additif : il est possible de réaliser des opérations sur des données chiffrées sans les déchiffrer, ce qui est extrêmement utile dans des domaines où la confidentialité prime.

La sécurité du système repose sur la difficulté du problème du résidu composite. Contrairement à RSA qui travaille modulo n, Paillier travaille modulo n^2, ce qui permet d'introduire de puissantes propriétés homomorphes.

2.6.1 Génération des clés

La génération des clés se passe comme ceci :

1. On choisit deux grands nombres premiers p et q tel que $gcd(pq, (p-1)(q-1)) = 1$.
2. On calcule $n = p \cdot q$ et $\lambda = lcm(p-1, q-1)$ où lcm indique le plus petit multiple commun (Least Commom Multiple).
3. On choisit un générateur $g \in \mathbb{Z}^*_{n^2}$.
4. On calcule $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$ avec $L(x) = \frac{x-1}{n}$. Pour calculer l'inverse modulo n, on utilise l'algorithme d'Euclide étendu commme dans RSA ce qui nous donne cette implémentation :

```
arg = expo_rapide(g, lambd, n_carre) #g^(lambd) mod n^2
L = (arg - 1) // n # Division entière

r, s, t = algo_Euclide(L, n) # calcule de l'inverse de L mod n
mu = s % n
```

On acquiert ainsi notre clé publique (n, g) et notre clé privée (λ, μ).

2.6.2 Chiffrement

Pour encrypter notre message m, il faut que celui-ci soit plus petit que n ($m < n$). On commence par générer un r tel que r est plus petit que n ($r < n$) et premier avec ce dernier.

```
r = random.randrange(1, n)
while math.gcd(r, n) != 1 :
    r = random.randrange(1, n)
```

Ensuit on peut simplement calculer notre message chiffré :

$$c = g^m \cdot r^n \mod n^2$$

2.6.3 Déchiffrement

Pour déchiffrer notre message encrypté, on va utiliser la même fonction $L(x) = \frac{x-1}{n}$ que lors de notre génération des clés pour calculer :

$$m = L(c^{\lambda} \mod n^2) \cdot \mu \mod n$$

2.6.4 Spécificité homomorphique

Comme pour le système de Benaloh, un message m encrypté avec le chiffrement de Paillier $c = \mathcal{E}(m) = g^m r^n \mod n^2$, le produit des valeurs chiffrées nous donnera la somme des valeurs d'origine :

$$\mathcal{E}(m_1) \cdot \mathcal{E}(m_2) = (g^{m_1} r_1^n) \cdot (g^{m_2} r_2^n) \mod n^2 = g^{m_1+m_2}(r_1 r_2)^n \mod n = \mathcal{E}(m_1 + m_2)$$

Comme précedemment, on aura une fonction pour effectuer cette addition :

```
def add_homomorphic(c1, c2, n) :
    return (c1 * c2) % (n**2)
```

En plus de cette possibilité d'addition, la cryptosytème de Paillier permet de multiplier une valeur chiffrée par une nombre non-chiffré. Bien que cette opération n'est pas entièrement chiffrée, elle reste une propriété intéressante de ce cryptosytème :

$$\mathcal{E}(m_1)^{m_2} \mod n^2 = \mathcal{E}(m_2)^{m_1} \mod n^2 = \mathcal{E}(m_1 m_2) \mod n$$

Nous avons donc une fonction qui effectuera ce produit mais contrairement aux autres, elle prend un nombre chiffré et un nombre clair comme argument :

```
def multiply_homomorphic(c1, m2, n) :
    return expo_rapide(c1, m2, n**2)
```

2.7 Librairie `TenSEAL`

`TenSEAL` est une bibliothèque Python permettant d'effectuer des calculs sur des données chiffrées à l'aide du chiffrement homomorphe. Elle repose sur la bibliothèque Microsoft SEAL et offre une interface simple pour le chiffrement, les opérations homomorphes, et le déchiffrement.

Elle permet notamment d'effectuer des opérations arithmétiques directement sur des données chiffrées, sans avoir à les déchiffrer au préalable, assurant ainsi la confidentialité tout au long du processus.

2.7.1 Initialisation

On initialise en premier un contexte de chiffrement qui sert de cadre de configuration cryptographique et dans lequel toutes nos opérations de chiffrement, déchiffrement et calculs homomorphes seront réalisées. On utilisera le schéma `BFV`, adapté aux entiers :

```
context = ts.context(
    ts.SCHEME_TYPE.BFV,
    poly_modulus_degree=8192,
    plain_modulus=786433,
    coeff_mod_bit_sizes=[60, 40, 40, 60]
)
context.generate_galois_keys()
context.generate_relin_keys()
```

2.7.2 Chiffrement

Les entiers sont chiffrés sous forme de liste pour permettre d'optimiser les opérations sur plusieurs d'entre eux en même temps. Dans notre essai, on va donc chiffrer deux valeurs dans deux listes distinctes :

```
encrypted_num1 = ts.bfv_vector(context, [num1])
encrypted_num2 = ts.bfv_vector(context, [num2])
```

2.7.3 Opérations homomorphes

Grâce au chiffrement avec `TenSEAL`, on peut très facilement effectuer des additions et des multiplications (des soustractions aussi mais nous ne nous y intéressons pas ici) sur nos valeurs encryptées de la même manière que si elles étaient au clair. Il faut juste faire attention car on fait nos opérations sur des listes et il faut donc que celles-ci soit de même dimension :

```
encrypted_sum = encrypted_num1 + encrypted_num2 # Addition homomorphe
encrypted_prod = encrypted_num1 * encrypted_num2 # Multiplication homomorphe
```

2.7.4 Déchiffrement

Pour déchiffrer nos valeurs, on utilise la méthode `decrypt()` de la librairie `TenSEAL` qui va effectuer le déchiffrement et nous donner le résultat, ce rendu sera aussi toujours sous forme de liste :

```
sum_decrypted = encrypted_sum.decrypt()
prod_decrypted = encrypted_prod.decrypt()
```

3 Résultats

Afin de pouvoir estimer nos cryptosystèmes, nous allons nous pencher sur deux aspects fondamentaux liés aux performances : le temps d'exécution et l'espace mémoire occupé.

3.1 Temps d'exécution

Le temps nécessaire pour effectué une opération est souvent négligeable mais lorsque d'innombrable calculs sont effectués, le temps que prend chacun d'entre eux peut prendre beaucoup d'importance. Dans notre implémentation, on peut relever ces valeurs :

Cryptosystème	Addition	Multiplication	XOR	Unité
Pas d'encryption	0.0	0.0	0.0	ms
RSA	-	1.0014	-	ms
ElGamal	-	0.9997	-	ms
Goldwasser–Micali	-	-	228.996	ms
Benaloh	0.9987	-	-	ms
Paillier	3.9999	4.0002	-	ms
TenSEAL	0.4508	29.8412	-	ms
Moyenne encryptée	1.8165	8.9606	228.996	ms

Table 1 – Tableau des temps d'exécution par cryptosytème.

Dans un premier temps, on remarque que, lorsqu'aucune encryption n'est présente, les opérations simples comme l'addition, la multiplication et le XOR ne prennent pas de temps (tellement petit que pas mesurable) à être effectuer.

En regardant le temps nécessaire pour une addition homomorphique, on voit que la librairie `TenSEAL` est particulièrement efficace en prennant moins d'une demi milliseconde pour effectuer le calcul. Le cryptosystème de Benaloh est légerement moins efficace mais effectue quand même l'opération assez rapidement. Le chiffrement de Paillier est nettement en dessous avec presque quatre millisecondes de temps d'exécution, ce temps plus important est compensé par la plus grande versatilité du système.

Pour la multiplication homomorphique, on a des résultats très similaire entre RSA et ElGamal. Comme pour l'addition, Paillier met environ quatre millisecondes ce qui reste plus long que les autres systèmes mais tout de même raisonnable. La plus grosse différence se fait avec le chiffrement de `TenSEAL` où l'opération prend quasiment trente millisecondes, cette encryption complèxe est visiblement plus optimisée pour des additions que pour des multiplications bien qu'elle garde son utilité multiple.

Le cryptosytème de Goldwasser–Micali met beaucoup plus de temps que tout le reste pour effectuer son XOR, ceci peut s'expliquer par le fait que le nombre est représenté sous forme de liste de bits et où un calcul est effectué pour chaque bit de ce dernier, le temps d'exécution est donc grandement multiplié et se voit fortement dans nos résultats.

3.2 Espace mémoire occupé

La mémoire, bien qu'aujourd'hui nous disposons facilement de grande capacité de stockage, reste un élément clé dans la recherche d'optimisation. C'est pourquoi nous avons pu relever ces résultats :

Cryptosystème	Taille du nombre	Après opération	Unité
Pas d'encryption	28	28	octets
RSA	92	92	octets
ElGamal	56	56	octets
Goldwasser–Micali	2208	2208	octets
Benaloh	92	160	octets
Paillier	160	160	octets
TenSEAL	334405	-	octets

TABLE 2 – Tableau de l'espace mémoire occupé par les nombres chiffrés par cryptosytème.

On remarque rapidement que la librairie `TenSEAL` prend un espace démesuré, son système est très nettement optimisé pour le temps d'exécution et de ce fait néglige la taille en mémoire. Le cryptosytème de Goldwasser–Micali est lui aussi significativement grand comparé aux autres méthodes, cette différence est principalement dû à notre manière de stocker sa valeur encrypté sous forme de liste de bits. La taille affichée étant uniquement la place occupée par la liste sans ses valeurs internes, elle est donc encore plus volumineuse mais peut être optimisé sur ce point en stockant des booléens au lieu d'"integer" pour la valeur de chaque bit. Une représentation sous forme de string aurait pu réduire l'espace occupé aussi.

Le chiffrement de ElGamal qui est le moins demandant en mémoire occupe déjà le double d'espace comparé aux chiffres non encrypté. On note aussi que plus le système est versatile, plus il occupe de mémoire avec Paillier qui occupe plus de place que RSA.

Benaloh est lui assez particulié étant donné que le nombre après l'opération homomorphique occupe plus de mémoire qu'avant. Cette différence peut s'expliquer par le fait que l'opération est une multiplication des deux nombres chiffrés, contrairement aux autres modèles cette multiplication n'est pas réduite au modulo n et peut donc occuper plus d'espace après opération.

4 Conclusion

Nous avons pu voir plusieurs méthodes de chiffrement homomorphique dans ce projet. En commençant par RSA et ElGamal qui étant relativement simple, ils se sont avérés efficace pour des opérations de multiplications sur des nombres encryptés, le modèle de ElGamal étant légerement meilleur que RSA sur tous les aspects homomorphiques. Le système de Goldwasser–Micali nous a permi d'effectuer des calculs sur chaque bits indépendamment mais voit ses performances chuter dû à cette spécificité. Ce dernier se voit utilisé dans le cryptosytème de Benaloh qui s'avère très efficace pour des additions homomorphes bien qu'il faille faire attention aux opérations successives au niveau de la mémoire. Le chiffrement de Paillier, quand à lui, s'est révélé moins performant que nos autres cryptosystèmes mais son utilité multiple est un argument non négligeable pour son utilisation.

Enfin, on a pu tester la librairie `TenSEAL` qui offre une possibilité de calculs homomorphes complets, optimisé pour effectuer plusieurs opérations en même temps et plus particulièrement pour les additions. L'espace mémoire occupé étant beaucoup plus grand que tous les autres sytèmes, il est quand même important de faire attention à ce niveau là.