Sécurité Bases de données – Rapport TP

Mise en place de systèmes de chiffrement dans une base de données

---

Rapport rédigé & produit par : **Kévin MOREAU & Florian VALLET**

Professeur responsable : **Sébastien GUILLET**

École Nationale Supérieure d'Ingénieurs de Bretagne Sud

Apprenti - Ingénieur Expert Cyberdéfense

2ème année

Promotion

AIRBUS

Table des matières

1. Rappel des objectifs

L'objectif de ce TP est d'étudier l'efficacité de deux types de chiffrements utilisés pour protéger des informations contenues dans une base de données classique.

La sécurité d'une base de données est quelque chose de relativement complexe à mettre en place. La sécurité doit assurer la confidentialité, l'intégrité et l'accessibilité relative des informations contenues dans la base.

L'action la plus importe pour sécuriser une base de données repose dans la sécurité en profondeur. En effet, le principe de base de la sécurité en profondeur réside dans le fait qu'on part du principe que la machine exécutant la base de données est corrompue. Malgré cela, il faut mettre en place l'ensemble des mesures de sécurité en profondeur pour conserver les principes de bases énumérés ci-dessus.

Ces mesures doivent être capable de lutter face à des attaques de type inférence de données.

Ce TP a pour but d'étudier ainsi l'une des mesures garantissant la confidentialité des données présentes en base de données : le chiffrement.

Deux types de chiffrements seront étudiés :

- Le chiffrement par relation d'ordre
- Le chiffrement homomorphique, avec une implémentation du cryptosystème de Paillier.

Ce TP répond aux questions 31 et 31bis suivantes :

Q31 : définissez une nouvelle base mysql avec une table dont une colonne contenant des entiers est chiffrée à l'aide d'un algorithme préservant la relation d'ordre — aka. Order Revealing Encryption — (afin que les requêtes sur intervalles soient permises), puis donnez l'implémentation (en python, c, ou java) du middleware et d'une application cliente de cette base illustrant la récupération et le déchiffrement des informations de la base ainsi que le bon fonctionnement de la relation d'ordre.

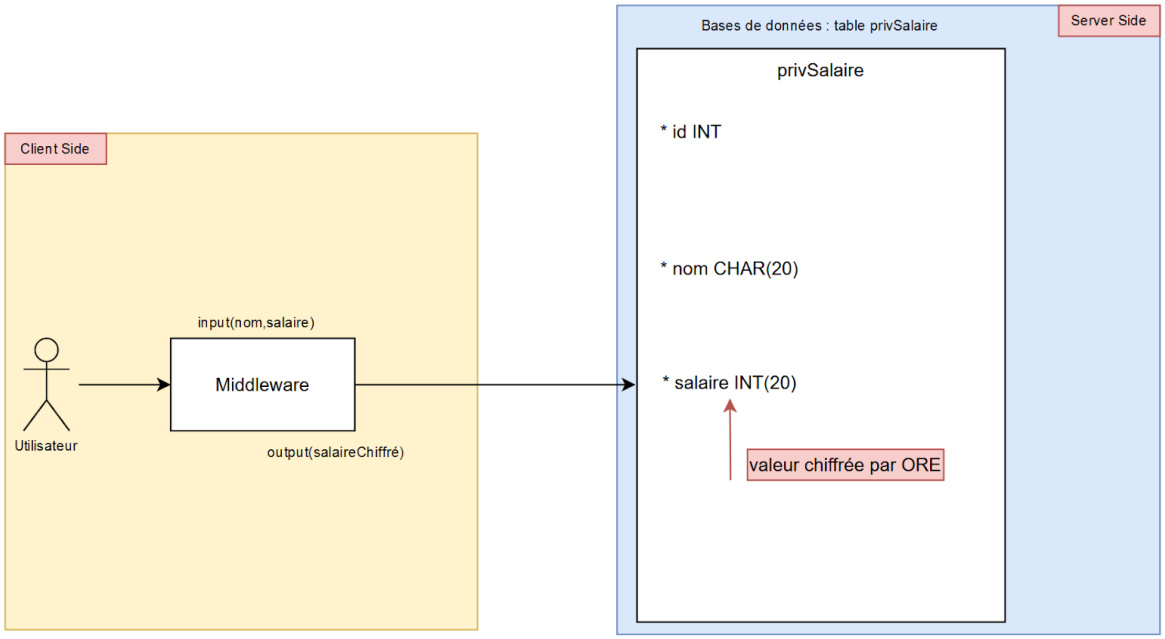
Focalisez-vous sur la preuve de fonctionnement, on n'attend pas ici un produit fini avec parseur de requête, une application cliente effectuant une requête sur intervalle, vérifiant la réponse, et terminant, est suffisant.

Q31bis : Implémenter un middleware côté serveur permettant un chiffrement homomorphique

---

2. Chiffrement ORE (relation d'ordre)

L'objectif de cette partie est de fournir une preuve de concept à travers un middleware répondant au principe de fonctionnement suivant, illustré sous forme de schéma :



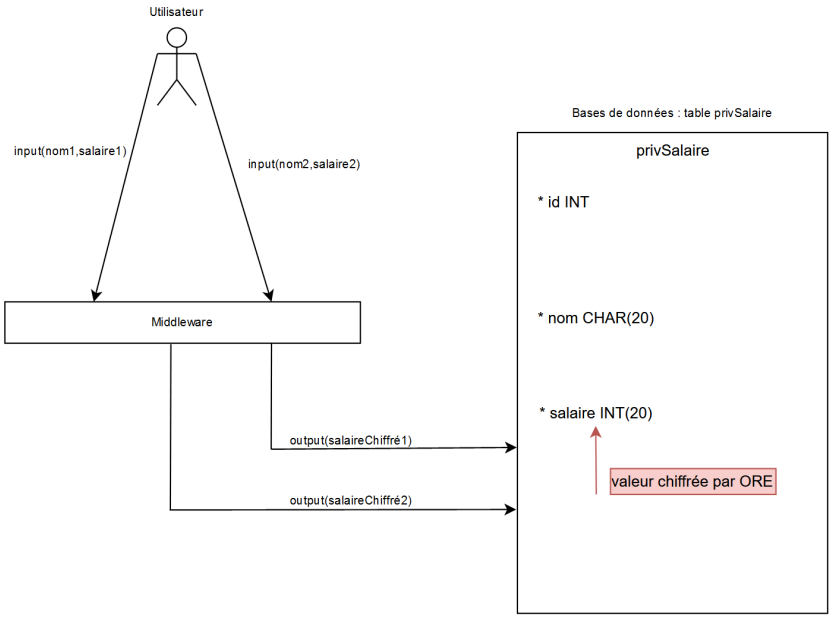
On souhaite montrer l'implémentation d'un chiffrement de type *ORE (Order Revealing Encryption)* permettant l'insertion de valeurs chiffrées au sein d'une table *privSalaire* contenant des salaires. Ces salaires seront donc stockés chiffrés. L'avantage d'un chiffrement *ORE* est qu'il conserve la relation d'ordre, ainsi, le *SGBD* pourra parfaitement répondre à des requêtes de type comparaison sur ce type de données sans se tromper, tout en garantissant la confidentialité des données lors d'une requête.

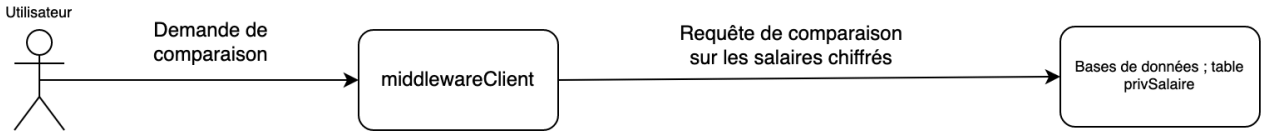
Pour permettre ce chiffrement, l'algorithme a besoin d'une clé. Par soucis de praticité, cette dernière sera stockée en dur dans le programme du *middleware*. Le but étant ici de mettre en avant une preuve de concept et non pas de fournir un programme 100% sécurisé.

Ce *middleware* étant implémenté côté client, il n'est pas possible de trouver la clé à partir du serveur.

Au niveau de l'utilisation, l'utilisateur mentionne le nom ainsi que la valeur du salaire à modifier en clair vers le middleware et ce dernier retourne la valeur chiffrée du salaire qu'il renseigne directement dans la base de données en fonction de la personne choisie.

L’implémentation finale suit ainsi le principe de fonctionnement suivant, confortant et affirmant le bon fonctionnement de la relation d’ordre lors de requêtes de comparaison :





Le programme créé permet différents types de fonctions :

```
Middleware v1 : Chiffrement par relation d'ordre :


    1. Afficher le contenu de la table de salaire (bdd)
    2. Q31 : Ajouter un salaire chiffre (ORE)
    3  Q31 : Dechiffrer le contenu de la base de donnee
    4. Q31 : Comparaison entre les salaires
    5. Q31bis : Executer un chiffrement conservant la capacite d'addition (Homomorphisme)
    6. Quitter

Que voulez vous faire ? (taper le chiffre) :
```

La première fonction permet l’affichage de la table de salaire. Pour rappel, les salaires contenus dans cette table sont chiffrés avec une clé stockée en dur dans le programme.

Voici un aperçu visuel de la table de données lors qu'une requête SQL classique depuis le client :

```
Middleware v1 : Chiffrement par relation d'ordre :


    1. Afficher le contenu de la table de salaire (bdd)
    2. Q31 : Ajouter un salaire chiffre (ORE)
    3  Q31 : Dechiffrer le contenu de la base de donnee
    4. Q31 : Comparaison entre les salaires
    5. Q31bis : Executer un chiffrement conservant la capacite d'addition (Homomorphisme)
    6. Quitter

Que voulez vous faire ? (taper le chiffre) : 1

Affichage du contenu de la base de donnees
(u'Neth', 133135740)
(u'Flo', 151568739)
(u'Laurent', 87179821)
(u'Adrien', 139631499)
```

Comme on peut le voir, les valeurs des salaires sont chiffrées tout en conservant la relation d'ordre. Par exemple, ici, on peut voir que le salaire de 'Flo' est plus élevé que celui de 'Adrien'.

On confirme ceci en déchiffrant les salaires contenus dans la base :

```
Affichage des salaires dechiffres
('Nom = ', u'Neth')
('Salaire = ', 2009)
('Nom = ', u'Flo')
('Salaire = ', 2312)
('Nom = ', u'Laurent')
('Salaire = ', 1321)
('Nom = ', u'Adrien')
('Salaire = ', 2112)
```

La relation d'ordre étant conservée, il est donc parfaitement possible d'effectuer des opérations de comparaisons pour voir qui, entre deux personnes, possède le salaire le plus élevé :

```
Middleware v1 : Chiffrement par relation d'ordre :


    1. Afficher le contenu de la table de salaire (bdd)
    2. Q31 : Ajouter un salaire chiffre (ORE)
    3  Q31 : Dechiffrer le contenu de la base de donnee
    4. Q31 : Comparaison entre les salaires
    5. Q31bis : Executer un chiffrement conservant la capacite d'addition (Homomorphisme)
    6. Quitter

Que voulez vous faire ? (taper le chiffre) : 4
Saisissez le nom de la premiere personne : Adrien
Saisissez le nom de la deuxieme personne : Flo
Connexion a la base de donnees...
Succes.

Le salaire de Flo est plus eleve que celui de Adrien
Preuve avec les salaires dechiffres :
Affichage des salaires dechiffre
('Nom = ', u'Neth')
('Salaire = ', 2009)
('Nom = ', u'Flo')
('Salaire = ', 2312)
('Nom = ', u'Laurent')
('Salaire = ', 1321)
('Nom = ', u'Adrien')
('Salaire = ', 2112)
```

D’un point de vue sécuritaire, cet algorithme garantit certes la confidentialité de la données, mais les comparaisons peuvent inférer des informations (telle personne à un salaire plus élevé que l’autre, mais moins que celle-ci, etc), permettant de s’approcher d’une valeur sans pour autant être 100% exact.

La conservation de ce genre d’informations sous cette forme n’est donc pas recommandée pour garantir pleinement la confidentialité d’une donnée. Les valeurs stockées dans la base de données ne doivent pas rendre possible ce problème d’inférence

Le chiffrement par relation d’ordre permet, comme on le voit à travers le programme lors de la comparaison des salaires, de conserver les opérations de comparaisons effectuées par le SGBD.

Le problème de ce type de chiffrement se pose sur les opérations mathématiques. En effet, ce type de chiffrement ne supporte pas les opérations mathématiques telles que l’addition et la multiplication.

Il est donc nécessaire de trouver un autre moyen d’effectuer ce genre d’opérations à travers le SGBD tout en garantissant de bout en bout la confidentialité des données (input, traitement, et résultat).

Pour cela, on va utiliser le chiffrement homomorphique permettant ce genre d’actions.

3. Chiffrement homomorphique (cryptosystème de Paillier)

Le chiffrement homomorphique que l'on a décidé d'implémenter est le cryptosystème de Paillier. Le principe de fonctionnement réside par la génération d'une paire de clé (publique et privée) suivant le principe de l'algorithme de cryptographie asymétrique RSA.

Calcul de la clé publique :

$$pk = N = p \,.\, q$$

Calcul de la clé privée :

$$sk = \varphi(N) = (p-1)\,.\,(q.-1)$$

Chiffrement d'un message :

Soit m un message à chiffrer avec $0 \leq m < N$. Soit r , un entier aléatoire tel que $0 < r < N$ (appelé l'aléa). Le chiffré est alors :

$$c = (1+N)^m \cdot r^N \mod N^2$$

Déchiffrement d'un message :

$$m = \frac{(c \cdot r^{-N} \mod N^2) - 1}{N}.$$

L'avantage de ce cryptosystème est qu'il est homomorphisme additif c'est à dire qu'avec la clé publique, un chiffré *c1 = Chiffrer(m1)* et un chiffré *c2 = Chiffrer(m2)*, il est possible de construire un chiffré *c3 = Chiffrer (m1 + m2)* sans pour autant connaître les messages *m1* et *m2* d'origine.

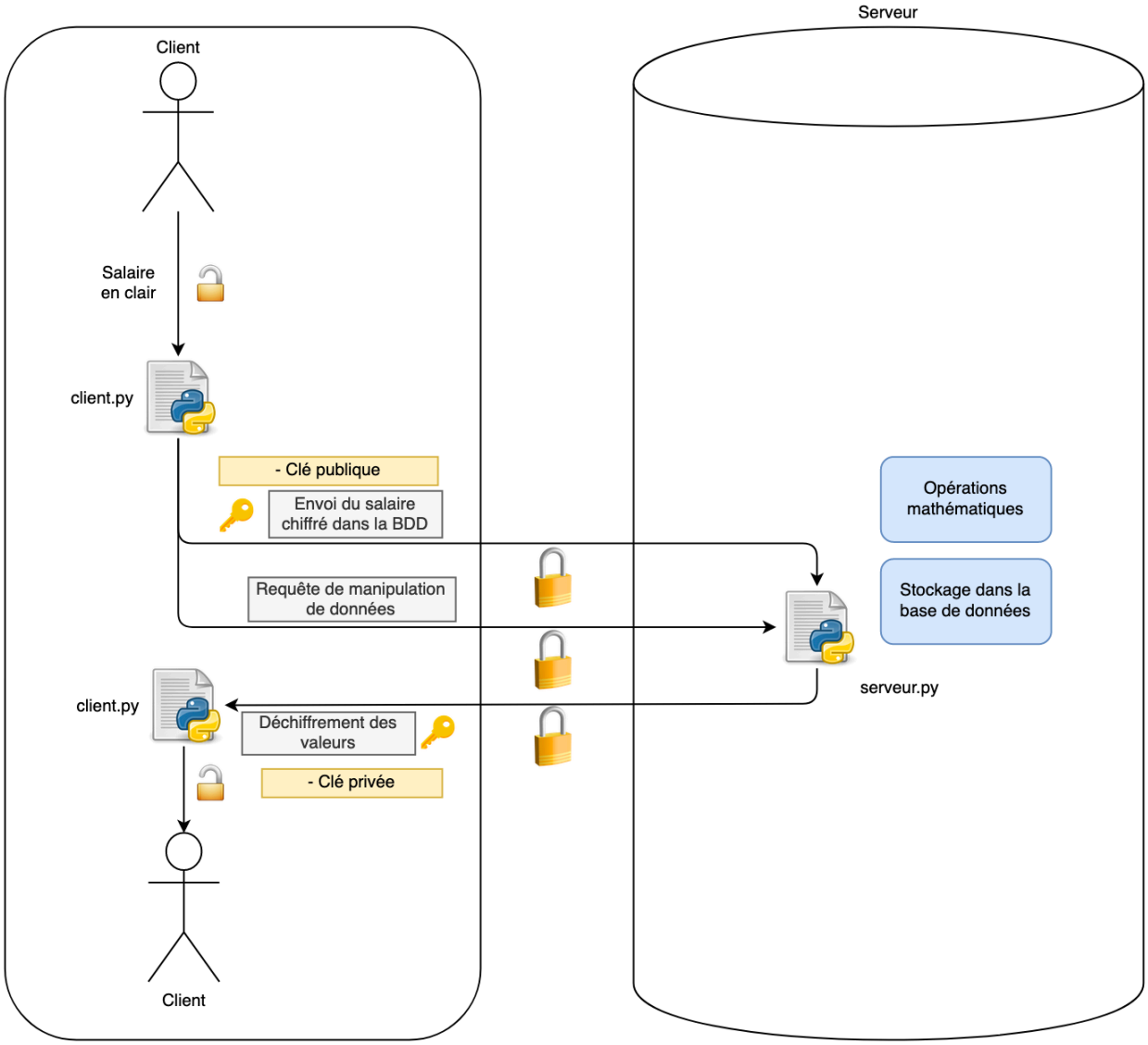
Voici la relation permettant ce phénomène :

$$\begin{aligned} c_1 \cdot c_2 &= (1+N)^{m_1} r_1^N \cdot (1+N)^{m_2} \cdot r_2^N \bmod N^2 \\ &= (1+N)^{m_1+m_2} (r_1 \cdot r_2)^N \bmod N^2 \end{aligned}$$

Qui correspond à un chiffré de $m_1 + m_2$ sous l'aléa $r_1 \cdot r_2$.

Concernant l'application sur ce TP, le chiffrement des messages se fera côté client. Le serveur manipule donc les données chiffrées et effectuera les opérations d'additions avant de renvoyer l'information vers l'utilisateur.

Ci-dessous un schéma explicatif de la relation client / serveur :



Ainsi, l'ensemble des données sera chiffré côté client dans le programme *client.py*. Le chiffrement homomorphique de ces données permettra au serveur d'effectuer les opérations mathématiques au sein du programme *serveur.py* et également de stocker / traiter des informations vers / depuis la base de données distante.

Côté environnement, 2 containers Docker seront utilisés :

- Le premier contient un serveur applicatif *python (flask)* où le programme *serveur.py* sera lancé automatiquement, et en écoute sur le port 5000.

- Le deuxième correspondant à la base de données *mysql*, écoutant les requêtes sur son port 3306.

Nous allons commencer notre étude sur le middleware côté client, intitulé *middlewareClient.py.*

Le chiffrement homomorphique commence par l'établissement de la paire de clés :
Pour faciliter la preuve de concept, on va générer une paire de clés faite maison avec nos propres valeurs de *p* et *q*.

Étant donné que nous sommes calibrés pour les implémentations de cryptosystème robustes, on se lance :

```
#Chiffrement PHE
public_key= paillier.PaillierPublicKey(2161831391)
private_key = paillier.PaillierPrivateKey(public_key, 47147,45853)
```

Les clés générés sont considérés comme des objets, ils possèdent donc des attributs et méthodes propres à ce type d'objet.

Les deux méthodes qui nous intéressent sont *encrypt* et *decrypt*.
Si ce dernier est utilisé à la fin du TP, c'est bien le premier sur lequel on va s'attarder au départ :

```
salairePHE= public_key.encrypt(salaire)
```

Cette méthode permet le chiffrement du salaire en clair (entré par l'utilisateur) en utilisant la clé publique, comme tout bon algorithme de chiffrement asymétrique.

En résulte un autre objet de type *EncryptedNumber*, possédant des caractéristiques intéressantes.

Pour pouvoir traiter cet objet, et pouvoir réaliser l'addition future, il est important de comprendre qu'est ce qui est généré derrière cet objet.

class `phe.paillier.`**`EncryptedNumber`**(*public_key*, *ciphertext*, *exponent=0*) [source]

Bases: **`object`**

Represents the Paillier encryption of a float or int.

Typically, an *EncryptedNumber* is created by **`PaillierPublicKey.encrypt()`**. You would only instantiate an *EncryptedNumber* manually if you are de-serializing a number someone else encrypted.

Paillier encryption is only defined for non-negative integers less than **`PaillierPublicKey.n`**. **`EncodedNumber`** provides an encoding scheme for floating point and signed integers that is compatible with the partially homomorphic properties of the Paillier cryptosystem:

1. D(E(a) * E(b)) = a + b
2. D(E(a)**b) = a * b

where *a* and *b* are ints or floats, *E* represents encoding then encryption, and *D* represents decryption then decoding.

Parameters:
- **public_key** (*PaillierPublicKey*) – the **`PaillierPublicKey`** against which the number was encrypted.
- **ciphertext** (*int*) – encrypted representation of the encoded number.
- **exponent** (*int*) – used by **`EncodedNumber`** to keep track of fixed precision. Usually negative.

Comme on peut le voir, l'objet généré est composé de 3 attributs étant la clé publique (*public_key*), le message chiffré(*ciphertext*) ainsi qu'un exposant(*exponent*).

Il est nécessaire de récupérer ces informations pour une utilisation future dans le middleware côté serveur.

C'est également cet objet qui sera stocké dans la base de données (une fois sérialisé).

Ainsi, le client doit communiquer ces informations jusqu'à la partie serveur à travers le réseau. Pour cela, nous allons utiliser l'une des composantes intéressantes de *python* qui permet de récupérer des informations au format *json* :

```
#Chiffrement PHE (le salaire en clair devient un objet PHE décomposé en 3 paramètres)
            salairePHE = public_key.encrypt(salaire)
            #Décomposition de la série composant l'objet salairePHE
            seriePHE = {'public_key': public_key.n, 'ciphertext': str(salairePHE.cipher-
text()), 'exponent': salairePHE.exponent}
            #Dump de la série pour exploitation par le serveur distant
            serializedPHE = json.dumps(seriePHE)
            #Préparation du payload
            #  Envoi vers serveur.py (nom de la personne + l'objet salaire décomposé sé-
rialisé)
            payload = {'nom':str(nom), 'salairePHE':serializedPHE}
            #   Envoi du payload vers le serveur d'écoute
            r = requests.post("http://0.0.0.0:5000/encrypted", json=payload)
```

salairePHE étant l'objet généré précédemment, il est maintenant aisé d'en récupérer les composantes et de créer une sérialisation qui sera ensuite dumpée au format *json*. Cette technique nous permet, par la suite, d'envoyer une requête à travers la fonction *requests.post()* directement vers notre serveur distant. Cette requête contenant ainsi notre chiffré sous la forme d'un objet sérialisé.

Côté serveur, l'implémentation est relativement simple, le serveur récupère la requête qu'il vient ensuite stocker dans une variable interne pour pouvoir manipuler ces informations en local :

```
@app.route('/encrypted', methods=['POST'])
def transfertEncryptedNumber():
    print(request.is_json)
    data= request.get_json()
    print(data)
    receivedEncrypted = data.get('salairePHE')
    print(receivedEncrypted)
```

Concernant la tête de notre requête, voici à quoi elle ressemble une fois postée sur le serveur flash distant, ici on visualise tout en bas ce qui nous intéresse, à savoir notre objet sérialisé (capture réalisée à l'époque où nous avions une paire de clé aléatoire, qui tenait le coup…) :

Par la suite, une fois notre objet récupéré côté serveur, on l'insère dans notre base de données à travers la fonction *updatePHEsalary()* :

```
def updatePHEsalary(nom, salairePHE):
    connec = db_connexion(db_user,db_password,db_host,db_db)
    cur = connec.cursor()
    query = "UPDATE salaire SET salairePHE=(%s) WHERE nom=(%s)"
    cur.execute(query, (salairePHE, nom))
    cur.close
    db_close(connec)
```

En visualisation la base de données, on obtient donc notre salaire stocké sous la forme d'un dictionnaire :

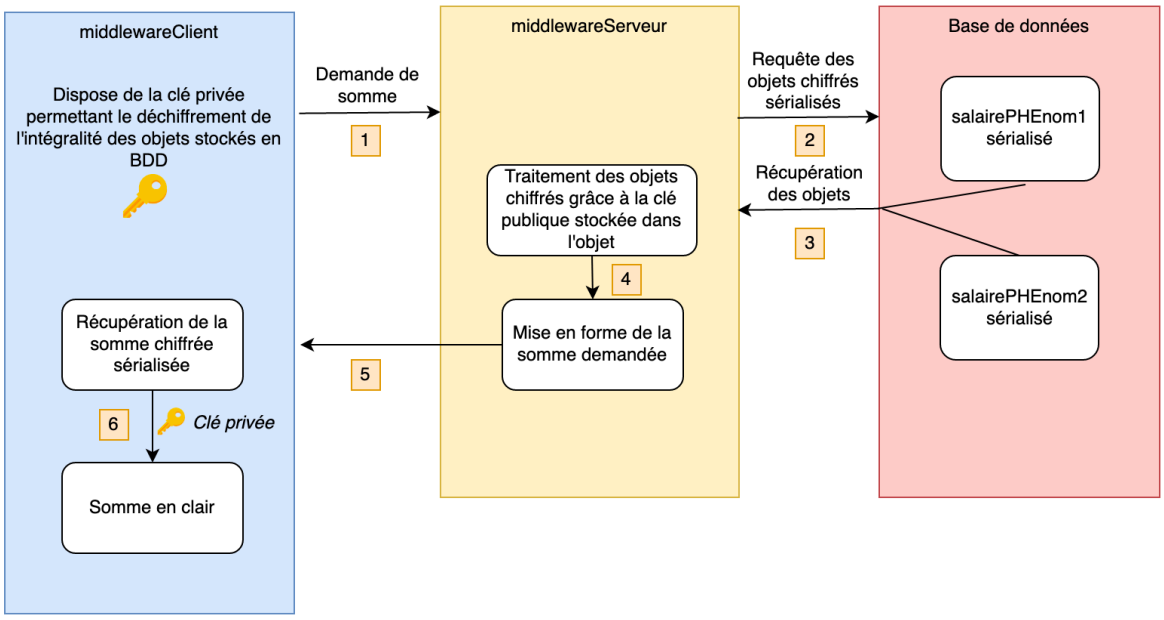
```
('MOREAU', 16465206, '{"public_key": 2161831391, "ciphertext": "1286692900320357518", "exponent": 0}')
('VALLET', 15532317, '{"public_key": 2161831391, "ciphertext": "952852186283463761", "exponent": 0}')
```

A partir de là, étant donné que la clé privée est stockée côté client, les données en base de données sont sécurisées (bien entendu, pas avec ce type de robustesse là).
Par contre, là où ce type de chiffrement devient intéressant, c'est que la clé publique est, quant à elle, directement stockée dans la base.

Comme on l'on vu, par définition, l'addition homomorphique n'ayant besoin que de la clé publique pour additionner deux chiffrés, toutes les données stockées en base peuvent être manipulées en garantissant la confidentialité du calcul.

En effet, comme nous allons le voir dans les prochaines requêtes, lors de la demande d'un calcul par le client, celles-ci viendront directement récupérer l'intégralité de l'objet requêté pour pouvoir effectuer une somme du côté du serveur.

Donc, cette fonctionnalité permet une protection optimale des données côté base de données car seul le middleware côté client, disposant de la clé privée, sera capable de déchiffré le contenu de l'objet.
La mise en pratique de la somme est détaillée juste après ce petit schéma récapitulatif en 6 étapes :



Côté client, on prépare un payload dans lequels sera stocké le nom de chacune des deux personnes que l'on souhaite ajouter (le salaire, of course..)

Le payload contient ainsi les informations suivantes :

```
nom1 = input('Saisir le nom de l\'employe 1 : ')
nom2 = input('Saisir le nom de l\'employe 2 : ')
payload = {'nom1':str(nom1), 'nom2':str(nom2)}
r = requests.post("http://0.0.0.0:5000/sum", json=payload)
```

Voici comment il est vu de la part du serveur lors de la requête *POST* :

```
True
{'nom1': 'MOREAU', 'nom2': 'DUPONT'}
```

Côté serveur, c'est ici que toute la partie de calcul va se faire :

```
def calculsomme(nom1, nom2):
    connec = db_connexion(db_user,db_password,db_host,db_db)
    cur = connec.cursor()
    query = "SELECT salairePHE FROM salaire WHERE nom=(%s) OR nom=(%s)"
    cur.execute(query, (nom1, nom2))

    somme = 0
    for salairePHE in cur:
        recuperationPHE = json.loads(salairePHE[0])
        n_of_Public_key = int(recuperationPHE['public_key'])
        public_key = paillier.PaillierPublicKey(n_of_Public_key=int(n_of_Public_key))
        valeurSalaire = paillier.EncryptedNumber(public_key, int(recuperation-
PHE['ciphertext']), int(recuperationPHE['exponent']))
        somme += valeurSalaire
    cur.close
    db_close(connec)

    serieSomme = {
        'public_key': public_key.n,
        'ciphertext': somme.ciphertext(),
        'exponent': somme.exponent
    }
    return serieSomme

def traitementSomme():
    data= request.get_json()
    print(data)
    nom1 = data.get('nom1')
    nom2 = data.get('nom2')
    print("Nom de la personne 1 : ",nom1)
    print("Nom de la personne 2 : ",nom2)
    result =  calculsomme(nom1,nom2)
    print("Affichage de la somme sérialisée : \n", result)
    seriesomme = calculsomme(nom1,nom2)
    return seriesomme
```

La fonction *traitementSomme()* va récupérer l'ensemble du *.json* sous la forme d'un dictionnaire. On associe ensuite chacun des noms requêtés à une variable propre qui seront passées en paramètre de la fonction *calculsomme().*

Cette dernière récupère les salaires associés aux noms insérés en paramètres à travers un curseur (*connector-mysql*).

Les deux salaires sont ensuite recomposés en objet *paillier.EncryptedNumber* grâce à ses attributs stockés : *public_key, ciphertext exposant*.

C'est ces objets qui sont au cœur du fonctionnement de l'addition homomorphique: cette dernière ne marchant uniquement que sur ce type d'objet.

Le résultat est ensuite re-sérialisé pour pouvoir être envoyé au client (*POST*), sous la forme d'un objet chiffré sérialisé.

Le client n'a plus qu'à *GET* l'objet et de le recomposer en objet *paillier.EncryptedNumber* avant de le déchiffrer dans les règles de l'art avec sa clé privée :

```
r = requests.get("http://0.0.0.0:5000/sumPost", json=payload).json()
            ##Reconstitution et Déchiffrement du message
            cipherSomme = int(r.get('ciphertext'))
            exponentSomme = int(r.get('exponent'))
            encryptedSommeObject = paillier.EncryptedNumber(public_key,cipherSomme,expo-
nentSomme)
            encryptedSommeCipher = private_key.decrypt(encryptedSommeObject)
            print("La somme des deux salaires est : ",encryptedSommeCipher)
```

Ainsi, le déchiffrement se faisant côté client, l'intégralité des échanges est confidentielle de bout en bout.

4. Sources & Bibliographie

- *phe.paillier — python-paillier 1.4.0 documentation*. https://python-paillier.readthedocs.io/en/stable/_modules/phe/paillier.html.
- « Python, How to Send data over TCP ». *Stack Overflow*, https://stackoverflow.com/questions/34653875/python-how-to-send-data-over-tcp
- « Python sending dictionary through TCP ». *Stack Overflow*, https://stackoverflow.com/questions/6341823/python-sending-dictionary-through-tcp
- KooR.fr. *KooR.fr - Fonction jsonify - module flask - Description de quelques librairies Python*. /Python/API/web/flask/jsonify.wp
- *Yet another Flask 405 error on POST and PUT : Forums : PythonAnywhere*. https://www.pythonanywhere.com/forums/topic/7695/
- antepher. « Flask: Parsing JSON Data ». *Techtutorialsx*, 7 janvier 2017, https://techtutorialsx.com/2017/01/07/flask-parsing-json-data/
- « python - Flask Value error view function did not return a response ». *Stack Overflow*, https://stackoverflow.com/questions/25034123/flask-value-error-view-function-did-not-return-a-response
- « python - Method Not Allowed flask error 405 ». *Stack Overflow*, https://stackoverflow.com/questions/21689364/method-not-allowed-flask-error-405
- *Formulaire d'une page Web*. https://pixees.fr/informatiquelycee/n_site/nsi_prem_flask.html
- « Containerize a Python App in 5 Minutes ». *Wintellect*, 20 décembre 2017, https://www.wintellect.com/containerize-python-app-5-minutes/
- « python - How to handle AttributeError: "NoneType" object has no attribute "get" in big dictionary ». *Stack Overflow*, https://stackoverflow.com/questions/21160181/how-to-handle-attributeerror-nonetype-object-has-no-attribute-get-in-big-di
- *API Documentation — python-paillier 1.4.0 documentation*. https://python-paillier.readthedocs.io/en/stable/phe.html
- *Requests: HTTP pour les humains — Requests 0.13.9 documentation*. https://requests-fr.readthedocs.io/en/latest
- « Python's Time.Sleep() - Pause, Stop, Wait or Sleep Your Python Code ». *Python Central*, 23 juillet 2013, https://www.pythoncentral.io/pythons-time-sleep-pause-wait-sleep-stop-your-code/
- « Error: "int" object is not subscriptable - Python ». *Stack Overflow*, https://stackoverflow.com/questions/8220702/error-int-object-is-not-subscriptable-python

- *SQL WHERE with AND, OR, and NOT | Examples*. https://www.dofactory.com/sql/where-and-or-not

- « python - <bound method Response.json of <Response [200]>> ». *Stack Overflow*, https://stackoverflow.com/questions/36240495/bound-method-response-json-of-response-200