



Rapport de Base de données

Implémentation de chiffrement pour la confidentialité de données

Introduction	2
Directive du Projet	2
Rapport : Implémentation technique	4
Conclusion	8

Introduction

Dans le cadre de ce projet, nous abordons deux problématiques essentielles en matière de sécurité et de confidentialité des données dans le contexte d'une base de données MySQL. La première consiste à concevoir une nouvelle base de données avec une colonne chiffrée, tout en préservant la relation d'ordre des entiers qu'elle contient. Pour ce faire, nous mettons en place un mécanisme de chiffrement qui permet non seulement de sécuriser les données, mais aussi de maintenir la possibilité d'effectuer des requêtes sur des intervalles, ce qui est une exigence fondamentale pour de nombreuses applications.

La seconde problématique s'articule autour de la confidentialité des données lors d'opérations effectuées à distance. Nous devons permettre à un serveur distant d'effectuer des opérations sur des données chiffrées sans pour autant révéler le contenu de ces données. Pour cela, nous devons concevoir un middleware côté serveur capable d'agir sur des données chiffrées, tout en préservant leur confidentialité.

Afin de répondre à ces défis, nous proposons une approche innovante reposant sur des techniques avancées de chiffrement et de manipulation de données. Dans un premier temps, nous élaborons une solution basée sur l'Order Revealing Encryption (ORE) pour chiffrer les données dans la base MySQL tout en permettant les opérations de comparaison sur les entiers chiffrés. Ensuite, nous développons un middleware côté serveur qui permet d'effectuer des opérations sur des données chiffrées tout en préservant leur confidentialité, en l'occurrence, une opération d'addition sur des entiers chiffrés.

L'objectif de ce rapport est de présenter notre approche, d'expliquer les choix techniques effectués pour chaque étape du processus de développement, d'illustrer la mise en œuvre de nos solutions à travers des exemples concrets, et enfin, de démontrer la validité et l'efficacité de notre approche à travers des tests et des analyses approfondies.

Directive du Projet

Consigne question 31

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, nous n'attendons pas ici un produit fini avec parseur de requête, mais une application cliente effectuant une requête sur intervalle, vérifiant la réponse, et terminant.

Consigne question 31bis

Nous souhaitons effectuer une opération à distance sur des données qui nous appartiennent, sans pour autant révéler à la machine distante le contenu de ces données. Implémentez un middleware côté serveur, capable d'effectuer une somme sur des données chiffrées par un client (le client est en mesure de déchiffrer cette somme). Combinez les deux middlewares afin que :

- il soit possible de comparer des entiers chiffrés (cf. relation d'ordre)
- il soit possible d'additionner des entiers chiffrés

Rapport : Implémentation technique

Architecture choisie

Notre architecture est basée sur Docker. Elle est composée d'un serveur MySQL, d'un script Middleware client, d'un script Middleware serveur et d'un script client. Voici les éléments techniques qui permettent de démarrer les services.

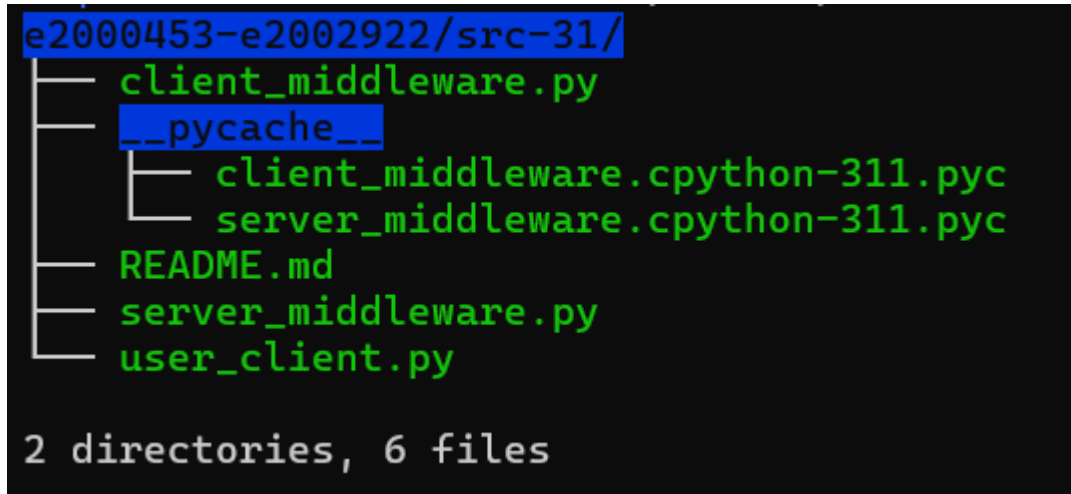


Fig n°1 : Arborescence du projet

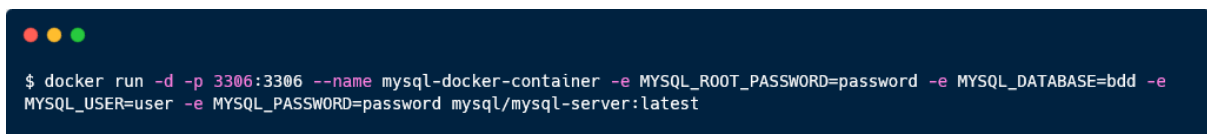


Fig n°2 : Mise en place du docker MySQL

Chiffrement ORE - Order Revealing Encryption

L'application Middleware client est donc une preuve de concept répondant au principe de fonctionnement suivant :

- Un utilisateur interagit, au travers du middleware, avec la base de données (ici permettant de stocker des personnes et leurs salaires au sein de la table "salaires_ope").
- La base de données stocke donc le nom de la personne (varchar(255)), et son salaire_encrypted (VARBINARY(255)) de façon chiffrée.

Nous souhaitons montrer l'implémentation d'un chiffrement de type ORE (Order Revealing Encryption) permettant l'insertion de valeurs chiffrées au sein d'une table "salaires_ope" contenant des salaires. Ces salaires seront donc stockés de manière chiffrée. L'avantage d'un chiffrement ORE est qu'il conserve la relation d'ordre, ainsi, le SGBD pourra parfaitement répondre à des requêtes de comparaison sur ce type de données, tout en garantissant leur confidentialité lors des requêtes. Afin de fonctionner, l'algorithme nécessite l'usage d'une clé symétrique. Dans notre cas, et par soucis de facilité, nous avons inscrit cette clé en

dur dans le code client du middleware. D'un point de vue utilisation, l'utilisateur précise le nom ainsi que le montant du salaire à modifier en clair, c'est ensuite transmis au middleware client qui va chiffrer la valeur et la transmettre à son tour au middleware serveur. Ce dernier va renseigner la valeur chiffrée du salaire qu'il renseigne directement dans la base de données en fonction de la personne choisie.

A ce stade, nous avons donc plusieurs fonctionnalités en place :

- Population de la base de donnée
- Afficher les salariés dans une range de salaire
- Afficher les salariés gagnant plus qu'une valeur
- Afficher les salariés gagnant moins qu'une valeur
- Insérer une nouvelle personne
- Comparer le plus haut salaire entre deux salariés

L'implémentation finale confirme le fonctionnement de la relation d'ordre lors de requêtes de comparaison.

Pour conclure cette implémentation d'algorithme ORE, nous pouvons dire qu'il permet de garantir la confidentialité des données, mais les comparaisons permettent tout de même de récupérer des informations importantes de notre table. En effet, nous pouvons connaître les relations d'ordre qui sont en vigueur dans notre jeu de données (telle personne est mieux payée que telle autre par exemple). 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.

De plus, un des problèmes de ce type de chiffrement se pose sur les opérations mathématiques : il 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 serveur MySQL tout en garantissant de bout en bout la confidentialité des données. Pour cela, nous allons utiliser le chiffrement homomorphique permettant ce genre d'actions.

Chiffrement Homomorphique - Implémentation de PyFHEL

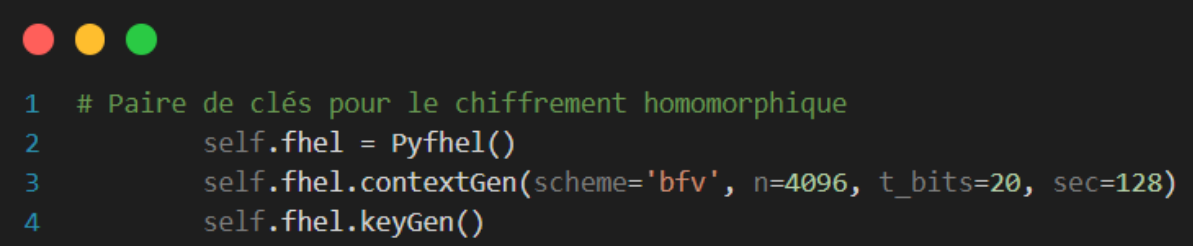
Pour cette seconde partie, nous avons décidé d'utiliser la librairie python PyFHEL mettant à notre disposition une implémentation de chiffrement homomorphique utilisant SEAL/PALISADE comme backend. Palisade étant lui-même une implémentation du cyprosystème Lattice. Le principe de cet algorithme reprend la logique de l'algorithme de cryptographie asymétrique RSA, en impliquant la génération d'une paire de clé publique/privée. Toutefois, contrairement aux algorithmes de chiffrement asymétrique classique, le cryptosystème Lattice est censé résister aux attaques utilisant l'algorithme de Shor.

Concernant le middleware mis en place pour répondre à notre besoin, nous devons adapter la partie client afin qu'il puisse s'interfacer avec notre middleware serveur. D'un point de vue fonctionnement, le chiffrement des données est réalisé par le middleware client puis envoyé au middleware serveur afin qu'il effectue les opérations et interactions avec le serveur MySQL. Le serveur reçoit alors des demandes de stockage, mise à jour, ou d'addition, qu'il traite avant de répondre au client. Finalement, le client déchiffre les réponses au besoin.

L'ensemble des données seront chiffrées côté client. Le chiffrement homomorphique des données permettra au serveur d'effectuer des opérations mathématiques et également de stocker et traiter des informations vers la base de données MySQL distante.

L'environnement Docker reprend celui présenté lors du chiffrement ORE, avec maintenant une table supplémentaire dans la base de données.

Le chiffrement homomorphique commence par l'établissement de la paire de clés, une nouvelle fois pour faciliter la preuve de concept, nous générons une paire de clés avec une valeur de n inscrite en dur dans le code.



```
1 # Paire de clés pour le chiffrement homomorphique
2 self.fhel = Pyfhel()
3 self.fhel.contextGen(scheme='bfv', n=4096, t_bits=20, sec=128)
4 self.fhel.keyGen()
```

Fig n°3: Mise en place de l'objet Cryptographique

La librairie PyFHEL retourne lors de la génération de clés des objets sur lesquels des attributs et méthodes sont disponibles. C'est justement certaines de ces méthodes propres à la librairie que nous allons exploiter par la suite : *encryptInt* et *decryptInt*

La première méthode est *encryptInt* :

```
1 # Fonction pour chiffrer un entier avec une clé publique
2 def chiffre_salaire_he(self, plaintext_salary):
3     # Homomorphic encryption using Pyfhel
4     encode_int = np.array([plaintext_salary], dtype=np.int64)
5     print(type(plaintext_salary))
6     int_ctx = self.fhel.encryptInt(encode_int)
7     encrypted_salary = int_ctx.to_bytes()
8     return encrypted_salary
```

Fig n°4 : Fonction de chiffrement Homomorphique

Elle permet le chiffrement du salaire en clair saisi par l'utilisateur, grâce à la clé publique. Un objet est retourné en sortie de cette méthode, il est converti en bytes (sérialisation de la donnée) afin de pouvoir le stocker dans la base de donnée dans une colonne avec l'attribut *LONGBLOB*. Ces informations sont essentielles au bon fonctionnement de notre middleware serveur, et ce sont elles que nous irons stocker dans notre base de données.

```
1 self.cursor.execute("""
2     CREATE TABLE IF NOT EXISTS salaires_he (
3         nom varchar(255) UNIQUE,
4         salaire_encrypted LONGBLOB
5     )
6 """)
```

Fig n°5 : Création de la table homomorphique

La fonction *get_encrypted_integers_from_db()* qui permet de récupérer les objets voulus dans la base de données et de calculer la somme chiffrée à retourner.

```
1 # Fonction pour récupérer les entiers chiffrés de la base de données
2 def get_encrypted_integers_from_db(self, nom1, nom2, hfel):
3
4     self.cursor.execute("SELECT salaire_encrypted FROM salaires_he WHERE nom=(%s) OR nom=(%s)", (nom1, nom2))
5     result = self.cursor.fetchall()
6     encrypted_integer_1 = PyCtxt(pyfhel=hfel, bytestring=bytes(result[0][0]))
7     encrypted_integer_2 = PyCtxt(pyfhel=hfel, bytestring=bytes(result[1][0]))
8     sum_result = encrypted_integer_1 + encrypted_integer_2
9
10    return sum_result
```

Fig n°6 : Fonction pour calculer la somme de deux salaires coté serveur

Enfin du côté client, nous récupérons la réponse mise à disposition par le middleware serveur.

```
1 def addition_data(self, nom1, nom2):
2     result = self.SrvMiddleware.get_encrypted_integers_from_db(nom1, nom2, self.fhel)
3     clear_salaire = self.dechiffre_salaire_he(result)
4     return clear_salaire[0]
```

Fig n°7 : Fonction pour calculer la somme côté client middleware

Finalement nous exploitons la méthode `dechiffre_salaire_he` afin de récupérer la valeur de la somme en clair.

```
1 # Fonction pour déchiffrer un entier avec une clé privée
2 def dechiffre_salaire_he(self, encrypted_int):
3     decrypted_int = self.fhel.decryptInt(encrypted_int)
4     return decrypted_int
```

Fig n°8 : Fonction pour déchiffrer la somme

Pour résumer, étant donné que la clé privée est stockée côté client, les données en base de données sont sécurisées. Comme observé, par définition, l'addition homomorphe n'ayant besoin que de la clé publique pour additionner deux données chiffrées, toutes les données stockées en base peuvent être manipulées en garantissant la confidentialité du calcul. En effet, comme nous pouvons le voir dans les interactions réalisées, lors de la demande d'un calcul par le client, les requêtes viendront directement récupérer l'intégralité de l'objet requêté pour pouvoir effectuer une somme côté serveur. Cette fonctionnalité permet alors une protection optimale côté base de données car seul le middleware client, disposant de la clé privée, sera capable de déchiffrer le contenu de l'objet.

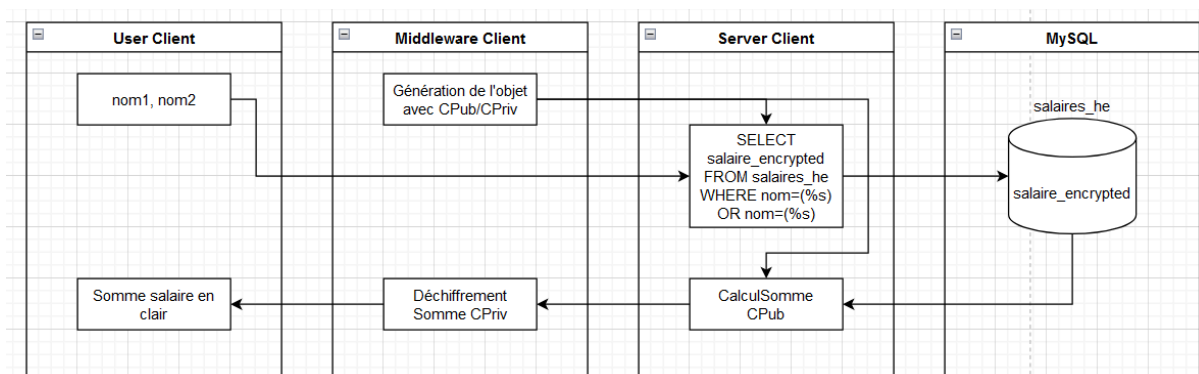


Fig n°9 : Process pour réaliser la somme de deux salaires

Conclusion

Pour conclure, nous pouvons dire que ce PoC impliquant du chiffrement homomorphique en base de données garantit la confidentialité des échanges et l'intégrité des données. Il faudrait néanmoins renforcer la robustesse des paires de clé en augmentant leur taille pour s'assurer qu'il ne soit pas possible de les casser dans un temps suffisamment court. Malheureusement, augmenter la taille des clés aurait un impact significatif sur les performances des applications sous-jacentes, et rendrait difficile l'usage en temps réel.