Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Hassiba Benbouali de Chlef

Faculté des Sciences Exactes & Informatique

Département d'informatique

Rapport

Tp Final

RESEAU ET SECURITE INFORMATIQUE

Sur: **AES**

Sous la direction de : Mr.Taher Abbas

Présenté par :

BOUZIANI SAID/GROUPE01/ISIA

BOUZIANE ERRAHMANI AKRAM/GROUPE01/ISIA

BOUZIANE MOHAMMED EL AMINE/GROUPE01/ISIA

Introduction

L'AES (Advanced Encryption Standard) est une méthode de cryptage largement utilisée pour sécuriser les données sensibles. Il utilise une clé de cryptage de longueur variable pour transformer les données en un texte chiffré, qui ne peut être lu sans la clé de décryptage correspondante. L'AES est considéré comme très sécurisé et est utilisé dans de nombreux domaines, y compris les communications sécurisées, les transactions bancaires et les systèmes de stockage de données.

Le principe de fonctionnement d'AES

1. **Ajout de clé (Add Round Key):** Dans cette étape, la clé de cryptage est combinée avec les données d'entrée via une opération XOR. Cette opération permet de mélanger la clé de cryptage avec les données et de les rendre plus complexes

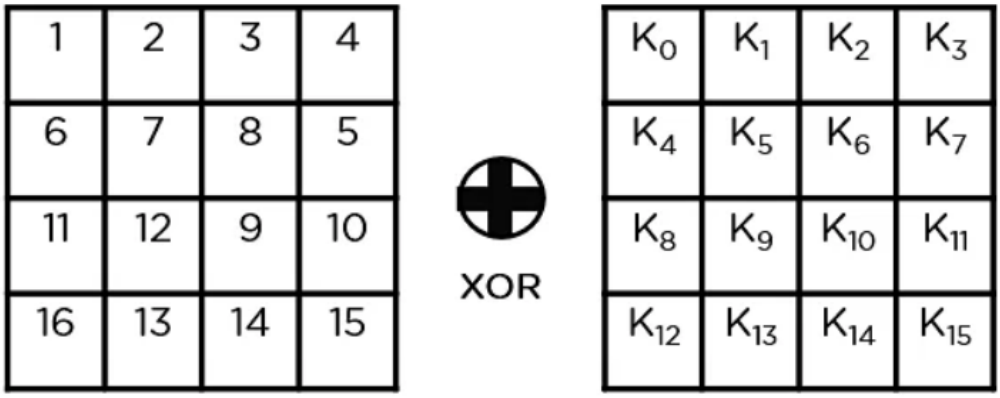

Figure 1:Add Round Key

2. **Substitution (Sub Bytes) :** Chaque octet dans les données est remplacé par un octet correspondant dans une table de substitution pré-définie, appelée S-Box. Cette table de substitution est basée sur une opération mathématique appelée inversion dans un corps fini (inverse dans un champ de Galois). La substitution de byte permet d'ajouter une couche de sécurité supplémentaire en masquant les données.

Figure 2:Sub-Bytes

3. **Décalage de ligne (Shift Rows) :** Les octets dans chaque ligne de données sont déplacés d'un certain nombre d'emplacements, en fonction de leur position dans la ligne. Ce décalage permet de mélanger les données et d'ajouter une autre couche de sécurité.

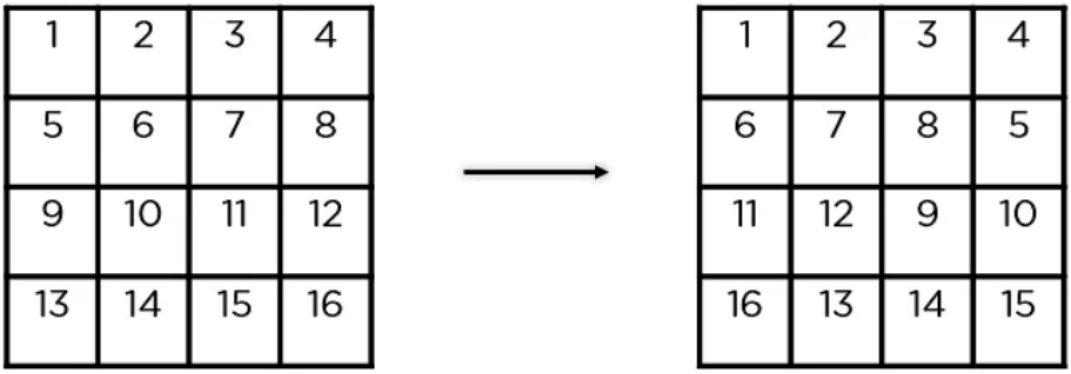

Figure 3: Shift-Rows

4. **Mélange de colonnes (Mix Columns) :** Les colonnes de données sont mélangées en utilisant une opération mathématique qui combine chaque octet avec les octets des autres colonnes. Cette opération permet de mélanger les données de manière plus complexe et d'ajouter une autre couche de sécurité.

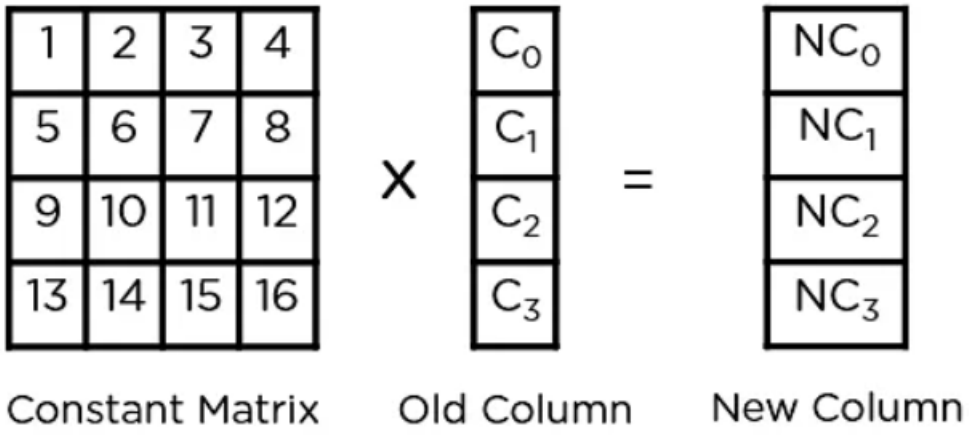

Figure 4: Mix-Columns

5. **Répétition des étapes 1 à 4 (Rounds) :** Les étapes 1 à 4 sont répétées plusieurs fois, en utilisant une nouvelle clé de cryptage dérivée de la clé originale à chaque tour. Le nombre de tours dépend de la longueur de la clé de cryptage utilisée. Plus la clé est longue, plus le nombre de tours est élevé.
6. **Dernier tour (Final Round) :** Le dernier tour est similaire aux tours précédents, mais la substitution de byte n'est pas effectuée, et le décalage de ligne est modifié pour éviter les collisions. Cela permet d'assurer que les données sont correctement déchiffrées lors du processus de décryptage.
7. **Ajout de clé finale (Final Add Round Key) :** La dernière clé de cryptage est combinée avec les données de sortie via une opération XOR pour produire le texte chiffré final. Cette opération permet de mélanger une dernière fois les données pour assurer leur sécurité.

Figure 5: Rounds

Le point fort d’AES :

Le point fort de l'AES est sa robustesse en termes de sécurité. Les différentes étapes de chiffrement (ajout de clé, substitution, décalage de ligne, mélange de colonnes) combinées avec la répétition de ces étapes plusieurs fois et l'utilisation de différentes clés de cryptage dérivées de la clé originale, rendent la tâche des attaquants extrêmement difficile. De plus, l'AES est largement utilisé et standardisé, ce qui facilite son implémentation et son utilisation dans de nombreux domaines.

Implementation en Python :

C: > Users > poupou > Desktop > test2.py > ...

```python
import os
from Crypto.Cipher import AES
import time

# La clé de chiffrement doit être de 16, 24 ou 32 octets de longueur
key = os.urandom(32)
# Initialisation du vecteur d'initialisation (IV) pour le mode CBC
iv = os.urandom(16)
# Création d'une instance de l'algorithme AES avec la clé et le mode CBC
cipher = AES.new(key, AES.MODE_CBC, iv)
# Le texte brut à chiffrer
plaintext = b"Hello, World! "
# Ajout de bourrage pour remplir les blocs de 128 bits
plaintext += b"\0" * (AES.block_size - len(plaintext) % AES.block_size)

# Measure execution time of encryption
start_time = time.perf_counter()
# Chiffrement du texte brut en utilisant l'algorithme AES
ciphertext = cipher.encrypt(plaintext)
end_time = time.perf_counter()
encryption_time = (end_time - start_time) * 1000

# Affichage du texte chiffré et du vecteur d'initialisation
print("Texte chiffré :", ciphertext.hex())
print("Vecteur d'initialisation :", iv.hex())

# Measure execution time of decryption
start_time = time.perf_counter()
# Déchiffrement du texte chiffré en utilisant la même clé et le même IV
decipher = AES.new(key, AES.MODE_CBC, iv)
decryptedtext = decipher.decrypt(ciphertext)
end_time = time.perf_counter()
decryption_time = (end_time - start_time) * 1000

# Suppression du bourrage ajouté
decryptedtext = decryptedtext.rstrip(b"\0")

# Affichage du texte déchiffré
print("Texte déchiffré :", decryptedtext.decode())

# Print execution times
print("Encryption time: {:.2f} ms".format(encryption_time))
print("Decryption time: {:.2f} ms".format(decryption_time))

```

Figure 6 Implementation AES sur python

Temps d'éxécution

- **Dans le cas de la clé à 16 Byte**

```
PS C:\Users\poupou> & C:/Users/poupou/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/poupou/Desktop/TP Final réseau/test2.py"
Texte chiffré : 79da33409acb473388362ddb8675f8aa
Vecteur d'initialisation : f574bf786d33eee67814f482ebc48587
Texte déchiffré : Hello, World!
Encryption time: 0.15 ms
Decryption time: 0.12 ms
PS C:\Users\poupou>
```

Figure 7 Temps d'éxecution, cas 16 byte

- **Dans le cas de la clé à 24 Byte**

```
PS C:\Users\poupou> & C:/Users/poupou/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/poupou/Desktop/TP Final réseau/test2.py"
Texte chiffré : d9f4056f5ae2d955fab81c68e5edb9b4
Vecteur d'initialisation : 147c70755a1b649e104c994aeb882390
Texte déchiffré : Hello, World!
Encryption time: 0.29 ms
Decryption time: 0.16 ms
```

Figure 8 Temps d'éxecution, cas 24 byte

- **Dans le cas de la clé à 32 Byte**

```
PS C:\Users\poupou> & C:/Users/poupou/AppData/Local/Programs/Python/Python311/python.exe "c:/Users/poupou/Desktop/TP Final réseau/test2.py"
Texte chiffré : ffe088385a611494f247bc826b86aa51
Vecteur d'initialisation : 58535d454d7864f476ef40c5de62b9d8
Texte déchiffré : Hello, World!
Encryption time: 0.33 ms
Decryption time: 0.21 ms
```

Figure 9 Temps d'éxecution, cas 32 byte

Implémentation en C++

```cpp
#include <iostream>
#include <chrono>
#include <cstring>
#include <openssl/aes.h>

using namespace std;

int main()
{
    // The encryption key must be 16, 24 or 32 bytes long
    // 16 Byte key c3795d5c5c5d3a6f525c6b63b1e617d9
    //24 Bytes key b67f2d2d558ee1017a520f1c94b6f3722149d9df3617f78a
    //32 Byte key 825a06ed7a8c1f1fbd5ab6dd9f6b8f6a5cf5d5dc5c3cb0771e03a193a0fc0bb3
    unsigned char key[] = "825a06ed7a8c1f1fbd5ab6dd9f6b8f6a5cf5d5dc5c3cb0771e03a193a0fc0bb3";
    unsigned char iv[] = "1234567890123456";

    // The plaintext to be encrypted
    unsigned char plaintext[] = "Hello, World!";

    // Padding the plaintext to fill the 128-bit blocks
    int plaintext_len = strlen((char*)plaintext);
    int padded_plaintext_len = ((plaintext_len + AES_BLOCK_SIZE) / AES_BLOCK_SIZE) * AES_BLOCK_SIZE;
    unsigned char* padded_plaintext = new unsigned char[padded_plaintext_len];
    memcpy(padded_plaintext, plaintext, plaintext_len);
    memset(padded_plaintext + plaintext_len, 0, padded_plaintext_len - plaintext_len);

    // Creating a new AES encryption context
    AES_KEY aes_key;
    AES_set_encrypt_key(key, 256, &aes_key);

    // Measuring encryption time
    auto start_time = chrono::high_resolution_clock::now();
    unsigned char* ciphertext = new unsigned char[padded_plaintext_len];
    AES_cbc_encrypt(padded_plaintext, ciphertext, padded_plaintext_len, &aes_key, iv, AES_ENCRYPT);
    auto end_time = chrono::high_resolution_clock::now();
    auto encryption_time = chrono::duration_cast<chrono::microseconds>(end_time -
start_time).count();
    // Displaying the encrypted ciphertext
    cout << "Ciphertext: ";
    for (int i = 0; i < padded_plaintext_len; i++) {
        printf("%02x", ciphertext[i]);
    }
    cout << endl;

    // Creating a new AES decryption context
    AES_KEY aes_key2;
    AES_set_decrypt_key(key, 256, &aes_key2);

    // Measuring decryption time
    start_time = chrono::high_resolution_clock::now();
    unsigned char* decryptedtext = new unsigned char[padded_plaintext_len];
    AES_cbc_encrypt(ciphertext, decryptedtext, padded_plaintext_len, &aes_key2, iv, AES_DECRYPT);
    end_time = chrono::high_resolution_clock::now();
    auto decryption_time = chrono::duration_cast<chrono::microseconds>(end_time -
start_time).count();
    // Removing padding from decrypted plaintext
    int original_plaintext_len = plaintext_len;
    while (decryptedtext[original_plaintext_len - 1] == '\0') {
        original_plaintext_len--;
    }

    // Displaying the decrypted plaintext
    cout << "Decrypted plaintext: " << decryptedtext << endl;

    // Displaying the execution times
    cout << "Encryption time: " << encryption_time << " microseconds" << endl;
    cout << "Decryption time: " << decryption_time << " microseconds" << endl;

    // Freeing memory
    delete[] padded_plaintext;
    delete[] ciphertext;
    delete[] decryptedtext;

    return 0;
}
```

Figure 10 Implémentation d'AES en C++

Temps d'éxécution

- **Clé à 16 Byte**

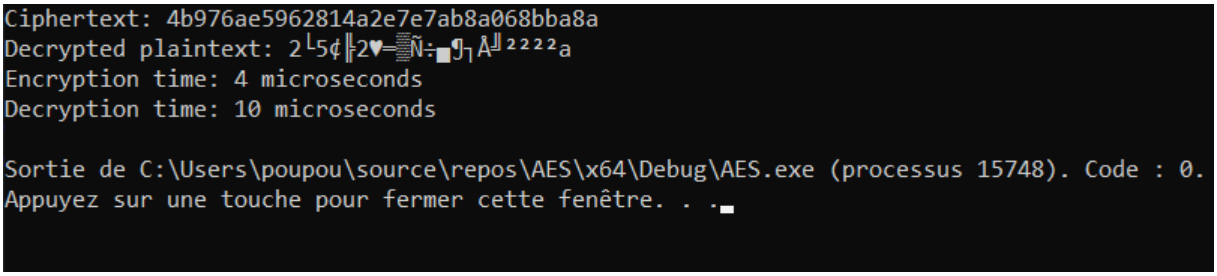

Figure 11 Temps d'éxécution , cas clé 16 Byte

- **Clé à 24 Byte**

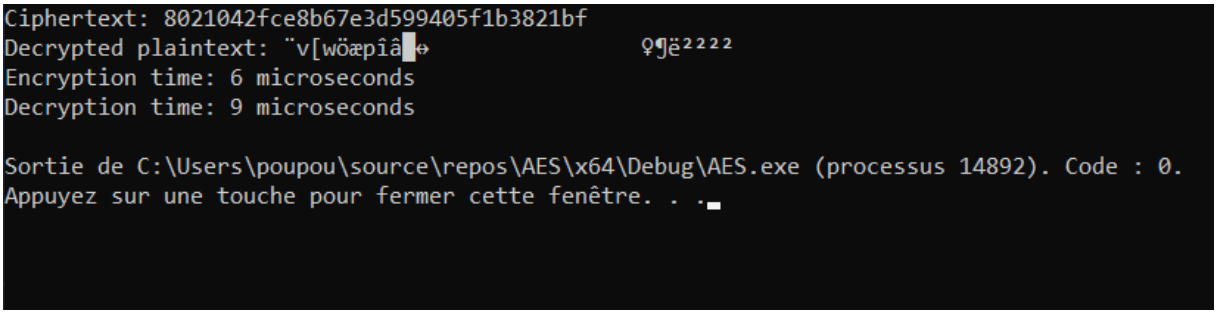

Figure 12 Temps d'éxécution , cas clé 24 Byte

- **Clé à 32 Byte**

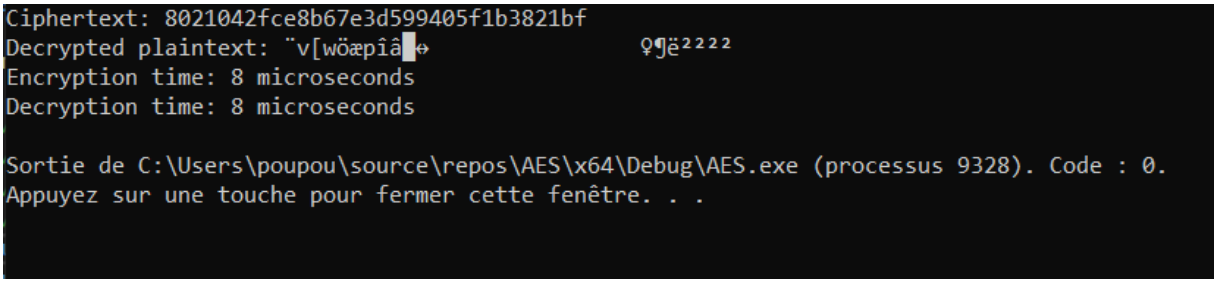

Figure 13 Temps d'éxécuttion, cas clé 32 Byte

Comparaison :

Les clés	Temps d'exe : **Python**	Temps d'exe : **C++**
Clé à 16 Byte	**Encryption :** 0.15 Milli s **Decryption :** 0.12 Milli s	**Encryption :** 0.004 Milli s **Decryption :** 0.01 Milli s
Clé à 24 Byte	**Encryption :** 0.29 Milli s **Decryption :** 0.16 Milli s	**Encryption :** 0.009 Milli s **Decryption :** 0.006 Milli s
Clé à 32 Byte	**Encryption :** 0.33 Milli s **Decryption :** 0.21 Milli s	**Encryption :** 0.008 Milli s **Decryption :** 0.008 Milli s

Figure 14: Tableau de comparaison entre python et c++ en terme de temps d'éxécution

Plot :

Les graphes ont été générés par la bibliothèque matplotlib de python :

```python
import matplotlib.pyplot as plt

x = [0.15, 0.29, 0.33]
z = [0.004, 0.009, 0.008]

a = [0.12, 0.16, 0.21]
b = [0.01, 0.006, 0.008]


y = [16, 24, 32]

plt.plot(y, x, color="red", label='Encryption: Python')
plt.plot(y, z, color="Green", label="Encryption C++")

plt.plot(y, a, color="Orange", label='Decryption: Python')
plt.plot(y, b, color="Blue", label="Decryption C++")


plt.legend()

plt.show()
```

Figure 15 Code de génération des graphes

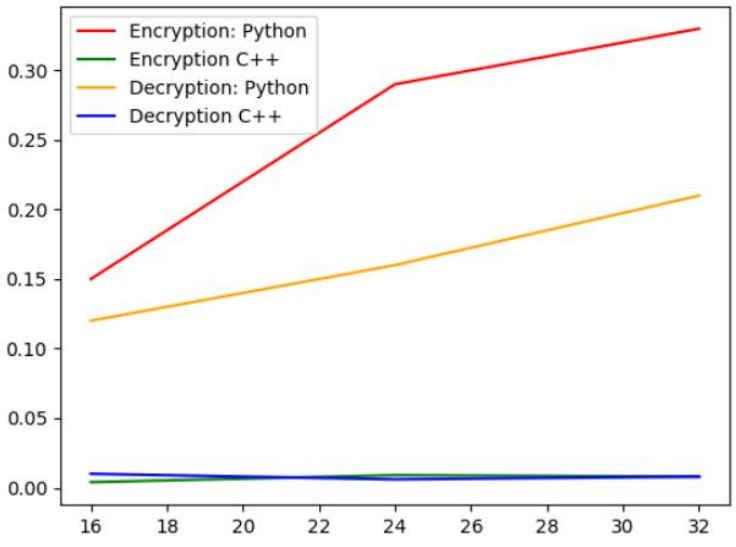

Figure 16 Graphe des temps d'éxécution de C++ et Python

Analyse du graphe :

Le tableau montre les temps d'exécution en millisecondes pour le chiffrement et le déchiffrement de données en utilisant l'algorithme AES avec des clés de 16, 24 et 32 octets. Les temps d'exécution sont comparés pour les langages Python et C++.

On peut remarquer que, pour toutes les tailles de clés, le temps d'exécution pour le chiffrement et le déchiffrement est nettement plus rapide en C++ qu'en Python. En particulier, pour la clé de 16 octets, le temps d'exécution pour le chiffrement est plus de 30 fois plus rapide en C++ que en Python. Pour la clé de 24 octets, le temps d'exécution est plus de 30 fois plus rapide en C++ pour le chiffrement et plus de 26 fois plus rapide pour le déchiffrement. Pour la clé de 32 octets, le temps d'exécution est plus de 40 fois plus rapide en C++ pour le chiffrement et plus de 26 fois plus rapide pour le déchiffrement.

C++ est généralement plus rapide que Python car il est compilé plutôt qu'interprété, ce qui signifie que le code est transformé en langage machine avant l'exécution. Python, d'autre part, est interprété, ce qui nécessite une interprétation à chaque exécution, ce qui ralentit le processus.

Les interfaces graphiques :

- **Interface D'encryption en python et PyQT :**

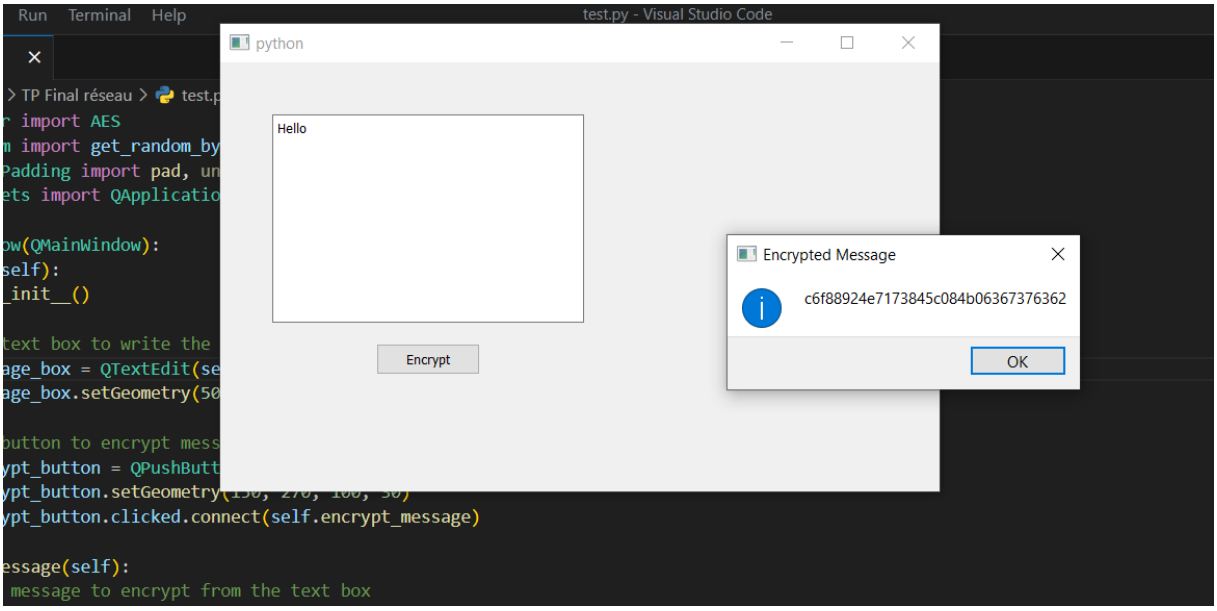

Figure 17: Interface 1

Cette Interface donne à l'utilsateur la main d'insérer un texte et le programme va en crypter et l'afficher

- **Inteface de communication avec cryptage des messages**

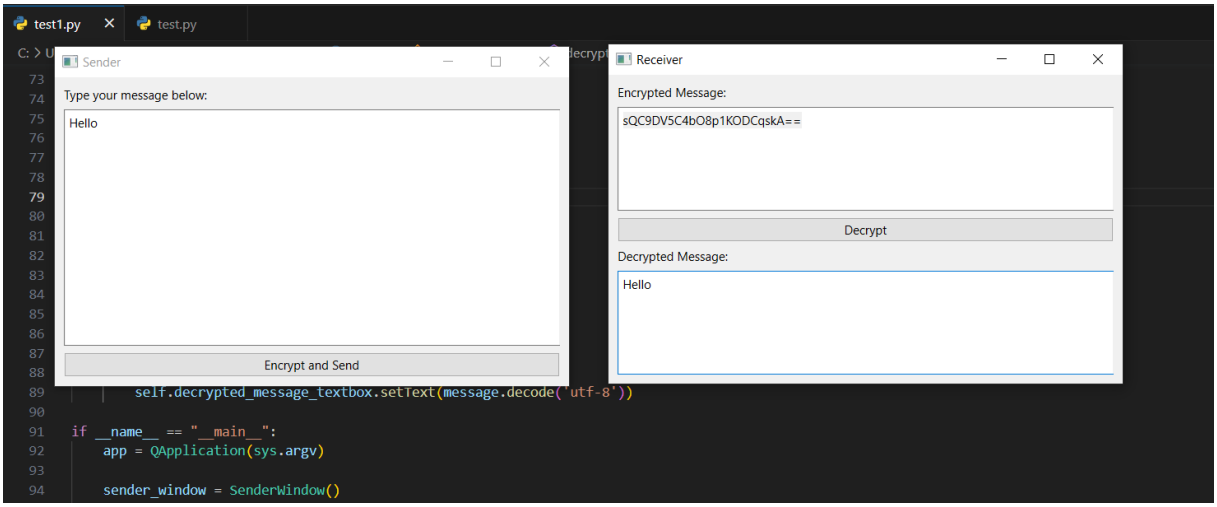

Figure 18 Interface 2

Cette interface représente la communication entre deux acteurs, « Sender » et « Receiver » .

- Le « sender » va écrire le message , le crypter et l'envoyer
- Le « reciever » va recevoir le message, le décrypter et l'afficher
- La communication a été réalisé grace aux « Slots » et « Signal » : **Les slots** sont des fonctions qui peuvent être appelées en réponse à un signal émis par un objet. Ils sont utilisés pour définir les actions qui doivent être effectuées lorsque des événements se produisent. **Les signaux** sont des événements émis par un objet lorsqu'une certaine action se produit. Ils permettent aux autres objets de s'inscrire pour recevoir des notifications lorsqu'un événement se produit.

Merci pour votre temps et considération !!