

Rapport SAE crypto - 2ème PARTIE

Membres

MARMION Steven SIMON Gael

Partie 1 : premières tentatives

En se basant sur les informations du sujet : La communication chiffrée entre Alice et Bob se base sur un chiffrement symétrique et asymétrique, en poursuivant l'analyse du sujet, on sait que :

- le chiffrement asymétrique est le RSA
- le chiffrement symétrique est inspiré de **DES** (SDES)

Ainsi, en connaissant le RSA, on sait que le RSA fonctionne de la manière suivante :

- On utilise une clé publique pour chiffrer et que l'on transmet avec le message chiffré
- On utilise une clé privée pour déchiffrer les messages reçues

Et pour faire le lien vers le sujet, ici notre clé publique échangée est notre **clé de session**. Notre clé de session étant un chiffrement symétrique et qui est **SDES**.

Ainsi, nous venons de résumer nos premières connaissances et les liens en fonction du sujet.

Répondez aux questions suivantes :

Question 1

En supposant que RSA soit utilisé correctement, Eve peut-elle espérer en venir à bout ? En vous appuyant sur votre cours, justifiez votre réponse.

Non, Eve ne pourra pas espérer en venir à bout car la condition même d'existence du RSA est la suivante : **"Une condition indispensable est qu'il soit « calculatoirement impossible » de déchiffrer à l'aide de la seule clé publique,"** [https://fr.wikipedia.org/wiki/Chiffrement_RSA].

Eve ne pourra donc pas espérer décrypter quelque chose car le temps qu'elle sera censée mettre pour décrypter pour avoir la clé privée sera beaucoup trop long et limite impossible. Le but du chiffrement RSA (ou tout autre chiffrement) est de rendre le temps de craquage beaucoup trop long et non rentable.

De plus, si l'on part du principe que RSA est utilisé correctement, on peut citer notamment le fait que l'on factorise de très grand nombre durant le processus et quand l'on factorise deux nombres, il est très difficile de trouver le chemin arrière :

► $p, q \rightarrow n = p.q$ facile (quadratique)

► $n = p.q \rightarrow p, q$ difficile

De plus, pour un chiffrement RSA et son déchiffrement, on instancie les relations suivantes :

Clé publique est (e, n) et la clé privée est d .

Les valeurs de puissance et les valeurs modulaires sont de simples exemple, elles n'ont aucun impact, le principal est de comprendre les relations et pourquoi RSA est un bon chiffrement

Chiffrer(T) :

$$\blacktriangleright (T^e) \bmod n = (T^{17}) \bmod 3233$$

Dechiffrer(C) :

$$\blacktriangleright (C^d) \bmod n = (C^{2753}) \bmod 3233$$

Où $d = e^{-1} \bmod \phi(n)$ sauf que $\phi(n) = (p - 1)(q - 1)$ et p et q sont secrets. Le seul indice que l'on a pour trouver p et q est n et $n = p \cdot q$. Ainsi, on revient au problème que nous avons énuméré tout à l'heure :

$$\blacktriangleright p, q \rightarrow n = p \cdot q \text{ facile (quadratique)}$$

$$\blacktriangleright n = p \cdot q \rightarrow p, q \text{ difficile}$$

Question 2

En quoi l'algorithme SDES est-il peu sécurisé ?

L'algo SDES utilise une clé de 10 bits pour chiffrer ce qui est peu $2^{10} = 1024$ combinaisons possible. il est donc possible de faire une attaque de force brute pour trouver la clé de chiffrement assez facilement.

Question 3

Est-ce que double SDES est-il vraiment plus sur? Quelle(s) information(s) supplémentaire(s) Eve doit-elle récupérer afin de pouvoir espérer venir à bout du double DES plus rapidement qu'avec un algorithme brutal? Décrivez cette méthode astucieuse et précisez le nombre d'essai pour trouver la clé.

En utilisant un double SDES cela permet de faire $2^{10} * 2^{10} = 1048576$ combinaisons possible. il est donc possible de faire une attaque de force brute pour trouver la clé de chiffrement mais cela prendra plus de temps donc oui le double SDES est plus sécurisé mais ce n'est qu'une question de temps, pas de réflexion. L'augmentation de la sécurité est donc subjective.

Pour que Eve puisse déchiffrer plus rapidement, elle aura besoin du principe meet-in-the-middle. Nous savons que le chiffrement utilisé se base sur une clé publique qui est partagée. Ainsi, nous savons que la clé

est utilisée pour chiffrer et déchiffrer les transactions de messages, ainsi, chiffrer l'envoi et déchiffrer le reçu. Nous pouvons donc nous baser sur le fait que nous devons simplement chiffrer une première fois et déchiffrer une première fois notre double chiffrement SDES et trouver la cohérence du premier chiffrement avec le premier déchiffrement. C'est ce que nous appelons le meet-in-the-middle. Si l'on résume cela, nous posons l'égalité suivante :

Nous partons du fait que le chiffré C s'obtient en faisant $ENC(K2, ENC(K1, M)) \Rightarrow SDES$

Par notre explication, nous obtenons ceci :

$$DEC(K2, C) = DEC(K2, ENC(K2, ENC(K1, M))) = ENC(K1, M)$$

A vous de coder

Chiffrement SDES

Le code actuel de SDES ne permet que d'encoder un seul bloc de 8 bits, soit 1 octet. De plus, le double chiffrement n'est pas fourni. Proposez une extension permettant de coder un texte quelconque, quelle que soit sa taille. Vous chiffrez notamment les textes disponibles sur Celene.

Ici, si on suit le sujet de la SAE, il est par logique décrit ceci :

le message est chiffré deux fois avec deux clés différentes:

C = SDES(SDES(M, k1), k2) Par principe, le code que l'on récupère ne fait que théoriquement **C = SDES(k2, message)** et nous nous avons besoin de **SDES(SDES(M, k1), k2)**, c'est-à-dire littéralement un double chiffrement. Ainsi, nous avons juste à doubler ce chiffrement donné, on répertorie donc le code suivant :

prérequis : nous avons mit SDES dans une classe et toutes ces fonctions avec.

Ici, on instancie une première fonction qui permet l'extension d'un double chiffrement comme le décrit SDES.

```
def double_chiffrement(self, cle1, cle2, message):
    """Encrypt twice the plaintext"""
    return self.encrypt(cle2, self.encrypt(cle1, message))
```

Ensuite, le but est de pouvoir étendre le chiffrement plus loin qu'un encodage de 8 bits, ainsi, on instancie le programme qui permet de coder n'importe quel texte :

```
def convertit_str_en_dec(message_en_str: str) -> list[int]:
    return list(ord(c) for c in message_en_str)
```

Ensuite, le but est de prendre la liste précédemment créée avec la fonction juste au-dessus et d'encrypter chaque lettre avec la méthode double_chiffrement pour SDES, ainsi, on peut poser le programme suivant :

```
def encrypt_tout_textes(self, plaintext: list[int], cle1, cle2):
    """Encrypt each character of the plaintext with one key"""
    res = []
    for dec in plaintext:
        res.append(self.double_chiffrement(cle1, cle2, dec))
    return res
```

Et finalement, nous obtenons cette liste là pour le fichier arsene_lupin_extraite.txt :

[124, 180, 203, 90, 244, 142, 90, 59, 90, 244, 142, 76, 1, 93, 142, 59, 254, 233, 59, 236, 93, 165, 76, 90, 59, 214, 19, 76, 225, 218, 252, 90, 59, 126, 233, 34, 93, 252, 144, 59, 159, 90, 252, 142, 236, 90, 89, 1, 252, 37, 242, 1, 89, 240, 76, 93, 253, 236, 90, 233, 76, 214, 59, 254, 90, 59, 60, 1, 233, 76, 93, 242, 90, 59, 126, 90, 240, 236, 1, 252, 242, 152, 124, 59, 193, 253, 233, 76, 242, 90, 107, 59, 236, 90, 59, 34, 76, 253, 248, 90, 142, 59, 83, 233, 142, 90, 252, 240, 90, 76, 159, 59, 55, 142, 142, 34, 225, 107, 189, 189, 8, 8, 8, 20, 159, 233, 142, 90, 252, 240, 90, 76, 159, 20, 253, 76, 159, 189, 90, 240, 253, 253, 85, 225, 189, 9, 132, 136, 145, 102, 152, 152, 19, 76, 225, 218, 252, 90, 59, 126, 233, 34, 93, 252, 59, 34, 1, 76, 89, 93, 59, 252, 253, 233, 225, 125, 59, 236, 71, 93, 252, 225, 1, 93, 225, 93, 225, 225, 1, 240, 236, 90, 59, 242, 1, 89, 240, 76, 93, 253, 236, 90, 233, 76, 59, 254, 253, 252, 142, 59, 253, 252, 59, 76, 1, 242, 253, 252, 142, 1, 93, 142, 152, 236, 90, 225, 59, 34, 76, 253, 233, 90, 225, 225, 90, 225, 59, 254, 1, 252, 225, 59, 142, 253, 233, 225, 59, 236, 90, 225, 59, 248, 253, 233, 76, 252, 1, 233, 244, 59, 254, 90, 34, 233, 93, 225, 59, 254, 90, 225, 59, 89, 253, 93, 225, 125, 59, 236, 71, 63, 252, 93, 159, 89, 1, 142, 93, 32, 233, 90, 152, 34, 90, 76, 225, 253, 252, 252, 1, 159, 90, 59, 1, 165, 90, 242, 59, 32, 233, 93, 59, 236, 90, 59, 165, 93, 90, 233, 244, 59, 83, 1, 252, 93, 89, 1, 76, 254, 144, 59, 252, 253, 142, 76, 90, 59, 89, 90, 93, 236, 236, 90, 233, 76, 59, 34, 253, 236, 93, 242, 93, 90, 76, 144, 59, 1, 165, 1, 93, 142, 152, 90, 252, 159, 1, 159, 63, 59, 242, 90, 59, 254, 233, 90, 236, 59, 200, 59, 89, 253, 76, 142, 59, 254, 253, 252, 142, 59, 236, 90, 225, 59, 34, 63, 76, 93, 34, 63, 142, 93, 90, 225, 59, 225, 90, 59, 254, 63, 76, 253, 233, 236, 1, 93, 90, 252, 142, 59, 254, 90, 59, 110, 1, 81, 253, 252, 59, 225, 93, 152, 34, 93, 142, 142, 253, 76, 90, 225, 32, 233, 90, 125, 59, 19, 76, 225, 218, 252, 90, 59, 126, 233, 34, 93, 252, 144, 59, 236, 90, 59, 110, 1, 252, 142, 1, 93, 225, 93, 225, 142, 90, 59, 159, 90, 252, 142, 236, 90, 89, 1, 252, 59, 32, 233, 93, 59, 252, 71, 253, 34, 218, 76, 90, 59, 32, 233, 90, 152, 254, 1, 252, 225, 59, 236, 90, 225, 59, 242, 55, 216, 142, 90, 1, 233, 244, 59, 90, 142, 59, 236, 90, 225, 59, 225, 1, 236, 253, 252, 225, 144, 59, 90, 142, 59, 32, 233, 93, 144, 59, 233, 252, 90, 59, 252, 233, 93, 142, 144, 59, 253, 181, 59, 93, 236, 59, 1, 165, 1, 93, 142, 59, 34, 63, 252, 63, 142, 76, 63, 152, 242, 55, 90, 5, 59, 236, 90, 59, 240, 1, 76, 253, 252, 59, 193, 242, 55, 253, 76, 89, 1, 252, 252, 144, 59, 90, 252, 59, 63, 142, 1, 93, 142, 59, 34, 1, 76, 142, 93, 59, 236, 90, 225, 59, 89, 1, 93, 252, 225, 59, 165, 93, 254, 90, 225, 59, 90, 142, 59, 1, 165, 1, 93, 142, 152, 236, 1, 93, 225, 225, 63, 59, 225, 1, 59, 242, 1, 76, 142, 90, 144, 59, 253, 76, 252, 63, 90, 59, 254, 90, 59, 242, 90, 142, 142, 90, 59, 110, 253, 76, 89, 233, 236, 90, 107, 59, 212, 19, 76, 225, 218, 252, 90, 59, 126, 233, 34, 93, 252, 144, 152, 159, 90, 252, 142, 236, 90, 89, 1, 252, 37, 242, 1, 89, 240, 76, 93, 253, 236, 90, 233, 76, 144, 59, 76, 90, 165, 93, 90, 252, 254, 76, 1, 59, 32, 233, 1, 252, 254, 59, 236, 90, 225, 59, 89, 90, 233, 240, 236, 90, 225, 59, 225, 90, 76, 253, 252, 142, 152, 1, 233, 142, 55, 90, 252, 142, 93, 32, 233, 90, 225, 41, 20, 59, 19, 76, 225, 218, 252, 90, 59, 126, 233, 34, 93, 252, 144, 59, 236, 71, 55, 253, 89, 89, 90, 59, 1, 233, 244, 59, 89, 93, 236, 236, 90, 59, 254, 63, 159, 233, 93, 225, 90, 89, 90, 252, 142, 225, 107, 59, 142, 253, 233, 76, 59, 200, 152, 142, 253, 233, 76, 59, 242, 55, 1, 233, 110, 110, 90, 233, 76, 144, 59, 142, 63, 252, 253, 76, 144, 59, 240, 253, 253, 85, 89, 1, 85, 90, 76, 144, 59, 110, 93, 236, 225, 59, 254, 90, 59, 110, 1, 89, 93, 236, 236, 90, 144, 59, 1, 254, 253, 236, 90, 225, 242, 90, 252, 142, 144, 152, 165, 93, 90, 93, 236, 236, 1, 76, 254, 144, 59, 242, 253, 89, 89, 93, 225, 37, 165, 253, 238, 1, 159, 90, 233,

```
76, 59, 89, 1, 76, 225, 90, 93, 236, 236, 1, 93, 225, 144, 59, 89, 63, 254, 90, 242, 93, 252, 59, 76, 233, 225,
225, 90, 144, 59, 142, 253, 76, 90, 76, 253, 152, 90, 225, 34, 1, 159, 252, 253, 236, 125]
```

En un temps considérablement petit : 0.00018072128295898438s

Proposez une méthode `cassage_brutal(message_clair,message_chiffre)` qui tente de retrouver les clés utilisées pour chiffrer le message en testant toutes les possibilités.

En partant du principe que les tailles de clés sont de 2^8 , on sait que le maximum d'une clé est 256, on peut donc poser le programme très simple mais très brutal suivant :

```
def cassage_brutal(sdes: DES.SDES, message_clair: list[int], message_chiffre:
list[int]):
    for i in range(constantes.LONGUEUR_CLE_BINAIRE):
        for j in range(constantes.LONGUEUR_CLE_BINAIRE):
            liste = sdes.encrypter_tout_textes(message_clair, i, j)
            if liste == message_chiffre:
                return i, j, liste
    return None, None
```

Ici, ce programme boucle deux fois pour retrouver les deux clés qui ont aidés à chiffrer le message, ici, comme en brut force, nous testons littéralement toutes les possibilités de clés.

Malheureusement, ce programme est un massacre, rien que pour déchiffrer le tuple de clés (5, 250), nous prenons 49sec. Nous imaginons pas pour un programme chiffrant les messages avec les clés (250, 250) ... Posons donc le tuple de clés (5, 250), nous prenons 49sec. Ainsi, pour faire une itération entière de la première boucle "for i in range(256):", nous prenons 49/5 sec, c'est-à-dire environ 10 sec par itération. On peut donc imaginer que pour le tuple de clés (250, 250), nous prenons $250 * 10\text{sec} \approx 0,6944444$ heure, c'est-à-dire environ 41,666664 min. Ainsi, le `cassage_brutal` est une catastrophe pourtant le programme est le plus simple possible.

Passons donc au `cassage_astucieux` :

Proposez une fonction `cassage_astucieux(message_clair,message_chiffre)` qui permettra de tester moins de possibilité de clés et ainsi réduire le temps d'exécution du cassage.

En partant du principe que l'on a une double boucle for. Si l'on voudrait faire moins d'itération. Nous pourrions faire une boucle for et une seconde mais qui ne sont pas imbriquées, du genre :

```
def cassage_astucieux(sdes: DES.SDES, message_clair: list[int], message_chiffre:
list[int]):
    dico=dict()
    for i in range(constantes.LONGUEUR_CLE_BINAIRE):
        res = []
        for dec in message_clair:
            res.append(sdes.encrypt(i,dec))
        dico[tuple(res)]=i # convertit la liste en tuple
    keys=dico.keys()
    keys=set(keys)
```

```

for y in range(constantes.LONGUEUR_CLE_BINAIRE):
    res = []
    for dec in message_chiffre:
        res.append(sdes.decrypt(y,dec))
    if tuple(res) in keys: # Convertit la liste en tuple
        return (dico[tuple(res)],y)
return False

```

Tout d'abord nous avons une premiere boucle qui va chiffrer le message claire avec une clé. La valeur ainsi obtenue est transformé en un tuple qui sera mis dans un dictionnaire avec comme clé le tuple et en valeur la clé de chiffrement. Ensuite nous avons une seconde boucle qui va déchiffrer le message chiffré avec une clé. Enfin on regarde si le tuple obtenu est dans l'ensemble des clés du dictionnaire, si il y a une correspondance on retourne la valeur liée au tuple (la clé 1) et la clé 2. C'est le principe de meet-in-the-middle expliqué tout à l'heure : trouver une cohérence entre le premier chiffrement et le premier déchiffrement.

Généralisation du nombre de tentative

Pour le texte d'arsene lupin, on expose sur le `cassage_brutal` et le `cassage_astucieux`, les temps suivants :

Pour le tuple de clés de chiffrement (5, 250) à retrouver par les deux méthodes brutal et astucieux :

`cassage_brutal` = **48.30299496650696 s** `cassage_astucieux` = **8.069215297698975 s**

Soit un ratio de 6.25. Nous avons donc un programme qui va 6 fois plus vite.

Pour le deuxième texte des lettres persanes, nous obtenons les temps suivants :

Pour le tuple de clés de chiffrement (10, 30) à retrouver :

`cassage_brutal` = **99.54146456718445 s** (1min39) `cassage_astucieux` = **5.428214073181152 s**

Pour la méthode `cassage_brutal` et le tuple de clé (5, 250). Nous pouvons simplement établir que le nombre de tentatives est la multiplication des deux clés. Puisque nous faisons une double boucle `for`, nous faisons donc pour le tuple (5, 250), 5 * 250 fois des itérations avant de trouver le bon tuple.

Pour la méthode `cassage_astucieux`, cela est mieux pensé. En sachant que nous ne faisons pas de double boucle `for` imbriqué mais plutôt deux boucles `for` l'une après l'autre, notre nombre de tentative baisse considérablement et nous savons que nous ne faisons plus `nb_tentative * nb_tentative` mais plutôt `nb_tentative + nb_tentative`.

Temps d'exécution

Comme dit juste avant, nous montrons par un seul essai que le `cassage_astucieux` est **nettement** plus rapide que le `cassage_brutal`. Toutefois, il est nécessaire de démontrer la moyenne du temps d'exécution des deux fonctions. Ainsi, nous allons poser la fonction suivante qui nous fait automatiquement l'expérience voulue sur un nombre de tentative entré en paramètre :

```

def fait_la_moyenne_des_temps(sdes, message_clair, message_chiffre,
    nombre_executions):
    total_tentatives_astucieux = 0

```

```

total_tentatives_brutal = 0
total_temps_astucieux = 0
total_temps_brutal = 0

for _ in range(nombre_executions):
    debut = time()
    tentatives_astucieux = cassage_astucieux(sdes, message_clair,
message_chiffre)
    fin = time()
    temps_astucieux = fin - debut

    debut = time()
    tentatives_brutal = cassage_brutal(sdes, message_clair, message_chiffre)
    fin = time()
    temps_brutal = fin - debut

    total_tentatives_astucieux += tentatives_astucieux[0] if
tentatives_astucieux else 0
    total_tentatives_brutal += tentatives_brutal[0] if tentatives_brutal else
0

    total_temps_astucieux += temps_astucieux
    total_temps_brutal += temps_brutal

moyenne_tentatives_astucieux = total_tentatives_astucieux / nombre_executions
moyenne_tentatives_brutal = total_tentatives_brutal / nombre_executions
moyenne_temps_astucieux = total_temps_astucieux / nombre_executions
moyenne_temps_brutal = total_temps_brutal / nombre_executions

return (moyenne_tentatives_astucieux, moyenne_temps_astucieux,
        moyenne_tentatives_brutal, moyenne_temps_brutal)

```

Et que nous appelons et affichons par les lignes suivantes :

```

resultats = fait_la_moyenne_des_temps(sdes, fichier_en_clair, fichier_chiffre, 10)
print("Moyenne de tentatives (astucieux):", resultats[0])
print("Moyenne de temps (astucieux):", resultats[1])
print("Moyenne de tentatives (brutal):", resultats[2])
print("Moyenne de temps (brutal):", resultats[3])

```

Et qui nous renvoie ceci :

```

Moyenne de tentatives (astucieux): 10.0
Moyenne de temps (astucieux): 5.944482040405274s
Moyenne de tentatives (brutal): 10.0
Moyenne de temps (brutal): 103.43257217407226s

```

Ici, nous voyons bien que pour une le cassage_astucieux nous mettons en moyenne 6sec et pour le cassage_brutal, nous mettons en moyenne 103 sec ce qui équivaut à 1min43. Nous avons donc un ratio de

$$103/6 = 17.$$

Nous en concluons donc que notre cassage_astucieux est 17 fois plus rapide que notre cassage_brutal sur une moyenne de 10 essais avec des clés inférieurs ou égales à 2^8 .

PARTIE 2 : Un peu d'aide

"A ce stade, Eve a bien travaillé mais se rend compte d'un problème: le protocole a été mis à jour et utilise maintenant l'algorithme AES avec des clés de taille 256 bits"

Question 1

Est-ce vraiment un problème ? Justifiez votre réponse.

Comme DES, AES est un chiffrement symétrique. Cela veut dire que la clé de chiffrement est utilisée pour chiffrer et déchiffrer. Ce chiffrement reste quand même un protocole compliqué à déchiffrer.

En prenant en compte que la taille des clés (avant) était de 8 bits et que là nous passons maintenant à une sécurité augmentée avec des clés de taille 256 bits. Nous pouvons répondre un grand OUI à cette question mais comprenons pourquoi ...

Pour nos programmes, pour trouver le tuple de clés, nous itérons sur une ou plusieurs boucles (en fonction du cassage brutal ou astucieux). Le problème c'est que nous itérons sur la borne maximale de la taille de la longueur de la clé et qui était, avant ce changement, une taille de 8 bits. Maintenant, si nous itérons sur une borne maximale de 2^{256} . Le temps va considérablement augmenté et nous pouvons réduire les chances pour Eve de trouver les deux clés proche de 0.

Le problème se trouve dans le temps que nous allons mettre à trouver les deux clés. En sachant que les clés qui ont chiffrés peuvent maintenant atteindre des grandeurs inimaginable, nous pouvons imaginer des clés telles que un décillion (1 suivi de 77 zéros) mais aussi des milliers, dizaine de milliers, des centaines de milliers, ... et nos temps d'exécution sur cassage_brutal sont typiquement exponentiel en sachant que nos tentatives sont la multiplication des deux clés donc par exemple le tuple (100 000, 100 000), nos tentatives seront du nombre de $100\,000 * 100\,000$.

Ainsi, ce changement de protocole nous mets une épine dans le pied.

Question 2

"Nous allons tenter d'illustrer expérimentalement les différences entre les deux protocoles. Vous évalueriez."

Question 2.1

Le temps d'exécution du chiffrement / déchiffrement d'un message avec chacun des deux protocoles. Ici vous devez le mesurer expérimentalement et donc fournir le code Python associé.

Partons du principe que l'on encrypte une seule fois avec AES...

Pour garder un point commun entre nos deux testes de protocole, nous allons chiffrer / déchiffrer le message compris dans le fichier **arsene_lupin_extrait.txt**.

*rappels : nous pouvons utiliser la bibliothèque cryptography ou Pycryptodome. Ici, nous utiliserons la bibliothèque **cryptography**.*

Ainsi, basons nos clés de chiffrement sur des chiffres qui sont bas. Ainsi, rien que par le temps d'exécution des chiffrements / déchiffrements de nos deux protocoles, nous verrons la différence de temps entre nos essais et pourront expérimentalement tenter d'illustrer les différences entre nos deux protocoles.

Posons donc le code qui chiffre AES suivant :

```
def chiffrer_AES(cle: int, message_a_chiffrer: str, vecteur_initilisation: str):
    cle = cle.to_bytes(32, byteorder='big') # Convertir la clé en bytes (256 bits
    ou 32 octets, taille des clés AES)
    mes_parametres = AES.new(cle, AES.MODE_CBC, vecteur_initilisation.encode('utf-
8'))

    # Ajouter un padding aux données pour les ajuster à une longueur multiple de
16 octets
    donnees_a_chiffrer = pad(message_a_chiffrer.encode('utf-8'), AES.block_size)

    texte_chiffre = mes_parametres.encrypt(donnees_a_chiffrer)

    return texte_chiffre
```

Et le code qui déchiffre AES suivant :

```
def dechiffrer_AES(cle: int, message_a_dechiffrer: bytes, vecteur_initilisation:
str):
    cle = cle.to_bytes(32, byteorder='big') # Convertir la clé en bytes (256
bits)
    mes_parametres = AES.new(cle, AES.MODE_CBC, vecteur_initilisation.encode('utf-
8'))

    donnees_dechiffrees = mes_parametres.decrypt(message_a_dechiffrer)

    # Retirer le padding après le déchiffrement
    donnees_originales = unpad(donnees_dechiffrees, AES.block_size).decode('utf-
8')

    return donnees_originales
```

En prenant en compte ces deux méthodes, nous allons pouvoir créer un programme qui récupère la clé utilisée pour le chiffrement et déchiffrement car rappelons que AES est un chiffrement symétrique, ce qui veut dire que la clé utilisée pour le chiffrement est la même que pour le déchiffrement.

Ainsi, on donne ce programme :

```
def cassage_AES_force_brut(message_chiffre: bytes, VI: str, message_dechiffre:
str):
```

```
for i in range(constantes.LONGUEUR_CLE_AES):
    try:
        texte_dechiffre = dechiffrer_AES(i, message_chiffre, VI)
        if texte_dechiffre == message_dechiffre:
            return i
    except Exception:
        pass
return None
```

Ici, nous allons faire notre protocole de teste.

Ainsi, on pose le programme suivant :

```
def temps_moyen_cassage_AES_force_brut(message_chiffre, VI, message_dechiffre):
    temps_total = 0
    essais = 10
    for _ in range(essais):
        debut = time.time()
        cassage_AES_force_brut(message_chiffre, VI, message_dechiffre)
        fin = time.time()
        temps_execution = fin - debut
        temps_total += temps_execution
    temps_moyen = temps_total / essais
    return temps_moyen
```

Ici, ce programme nous retourne la valeur suivante :

```
Temps moyen pour le cassage par force brute pour AES : 0.0022826433181762696 sec
pour 10 essais
```

Ici, on peut voir que nos programme ne sont pas comparable, d'un côté, on a un cassage brutal pour SDES qui prend 1min30 en moyenne et de l'autre côté un temps moyen de 0.002 sec pour le même texte concernant AES.

Ainsi, on sait qu'on change de protocole mais pas de manière de chiffrer. Ce qui veut dire que de la même manière que pour SDES, nous allons chiffrer deux fois notre message. Ainsi, on se retrouve avec ce programme :

Question 2.2

Le temps de cassage d'AES (même pour un cassage astucieux) si vous deviez l'exécuter sur votre ordinateur. Ici il faut uniquement estimer le temps nécessaire (sinon vous ne pourriez pas rendre votre rapport à temps!). Vous préciserez votre configuration et vous fournirez le détail des calculs.

Pour illustrer les calculs, nous pouvons poser le contexte suivant :

Nous savons que AES utilise des clés de 256 bits, ce qui signifie qu'il y a 2^{256} clés possibles. Nous prenons ensuite, dans notre contexte, des ordinateurs extrêmement puissants capables de tester des milliards de clés par seconde. Pour calculer notre estimation, nous supposons que l'ordinateur teste donc un milliard de clés par seconde :

$$2^{256} \text{ clés} / 10^9 \text{ clés par seconde} = 2^{256} / 10^9 \text{ secondes}$$

2^{256} étant donc la taille des clés utilisée par AES et 10^9 le nombre de clés testées par seconde de notre ordinateur.

Ce qui équivaut à 10^{75} secondes. Pour nous illustrer cette durée, convertissons la en années :

$$10^{75} \text{ secondes} / (60 \text{ secondes} * 60 \text{ minutes} * 24 \text{ heures} * 365 \text{ jours}) \sim 3 * 10^{67} \text{ années}$$

Ici, on remarque donc que même avec des capacités de PC très très optimistes, le cassage d'AES reste totalement inatteignable avec les ressources technologiques actuelles.

Evidemment, nous prenons ces estimations et ce contexte face à une attaque de force brut.

Question 3

"Il existe d'autres types d'attaques que de tester les différentes possibilités de clés. Lesquelles? Vous donnerez une explication succincte de l'une d'elles"

Parmi les types d'attaques, nous pouvons retrouver celles-ci, l'« Analyse Différentielle ».

[https://fr.wikipedia.org/wiki/Cryptanalyse_diff%C3%A9rentielle]

L'attaque par Analyse Différentielle. Cette attaque repose sur l'observation du comportement de l'algorithme lorsque des entrées légèrement différentes sont chiffrées. L'idée est d'analyser comment de petites différences dans les entrées affectent les sorties chiffrées. En répétant ce processus de chiffrement avec différentes entrées et en observant les différences dans les sorties, un attaquant peut déduire des informations sur la clé secrète ou sur le fonctionnement interne de l'algorithme.

Pour citer d'autres attaques, nous pouvons citer celle-ci qui est très minutieuse et qui très détaillée et assez compliqué à analyser: [https://fr.wikipedia.org/wiki/Attaque_par_canal_auxiliaire]

Le "Side-Channel". Contrairement à une attaque qui cherche à briser directement le processus de chiffrement, une attaque par canal auxiliaire exploite les informations obtenues à partir d'un système physique qui entoure l'exécution de l'algorithme de chiffrement. Par exemple, une attaque par canal auxiliaire peut se concentrer sur la consommation d'énergie, les temps de réponse, les émissions électromagnétiques ou même les sons émis par un dispositif lorsqu'il effectue des opérations de chiffrement. En analysant ces éléments, un attaquant peut déduire des informations sur les clés utilisées ou les données elles-mêmes, même sans accéder directement à l'algorithme de chiffrement ou à ses sorties.

Question 4

contexte : Eve décide de demander de l'aide à son ami MrRobot, célèbre hacker. Par un miracle cryptographique inexplicable (ou une terrible négligence de nos deux amoureux), il arrive à mettre la main sur la clé utilisée par Alice et Bob. Malheureusement, son état mental n'étant pas toujours très stable, il est rare qu'il fournisse une

réponse simple mais plutôt des énigmes à résoudre. Il vous envoie donc deux images, d'apparence identique... Cependant il ne faut pas se fier aux apparences... Récupérez sur Celene les deux images proposées et analysez-les afin de trouver la clé. A vous de jouer ! Vous présenterez dans votre rapport non seulement la solution de l'énigme mais également le détail des étapes qui vous ont permis de la résoudre. Si vous avez dû implémenter du code python, il faudra le fournir.

Pour résoudre ce problème, nous avons posé notre problème selon deux questions :

Comment une image pourrait-elle cacher des informations ?

Et

Comment pourrions-nous voir le message caché ?

Pour répondre à ces questions, nous avons tout d'abord réfléchi sur l'indice premier de l'énoncé, nous devons trouver un message dans les deux images... Par logique, nous avons donc commencés par reprendre les bases de nos connaissances sur les images, c'est-à-dire, leur consitions. Nous savons qu'une image est constitué de pixels, nous savons de-même qu'il existe des outils qui permettent de regarder des images dans leurs détails en détaillant leurs en-têtes, leurs versions, etc ... ainsi que les valeurs de leurs pixels.

Pour commencer, nous avons donc utiliser ces outils en se focalisant sur les détails secondaires d'une image, c'est-à-dire toutes ces données hors pixels .Malheureusement, après beaucoup de temps passé sur ces données, nous n'avions rien trouver de très intéressant. C'est pour cela que nous avons repris le sujet et décortiquer les indices. Sur cette base, nous avons identifié un indice qui était intéressant et que nous avons pas pris en compte en premier lieu : [...] *Il vous envoie donc deux images, d'apparence identique... Cependant il ne faut pas se fier aux apparences...*

De cette indice, nous en avons compris que le message caché dans ces messages ne pouvait se trouver que dans leurs apparences et rappelons que l'apparence d'une image se forme grâce aux pixels qui la constitue !

Ainsi, nous sommes partis sur cette deuxième base, ne pas chercher les éléments secondaires comme les en-têtes, etc ... mais chercher dans les éléments primaires d'une image, ses pixels.

Ainsi, nous avons répondu, de par cette réflexion, à notre première problématique : **Comment une image pourrait-elle cacher des informations ?**

Et la réponse ne pouvait être que dans ces pixels.

Malgré le fait que nous avons trouvé comment le message pouvait potentiellement se cacher, nous devons savoir comment un message pouvait se cacher dans deux images différentes...

Pour cela, nous nous sommes questionnés de la manière suivante :

Nous savons que le message se trouve dans les deux images et que nous ne devons pas nous fier aux apparences. Ainsi, la première étape a été de visualiser les deux images. Elles sont identiques... Pourtant, nous avons résonnés tout à l'heure sur les pixels qui pouvait potentiellement cacher un message. Ainsi, on pose le premier programme suivant qui permet d'afficher tout les pixels d'une image pour voir ce que nous pouvons en tirer.

```
for image in liste_images_a_traiter:
    i = open(image)
```


```
if (image==IMAGE_ROSSIGNOL1):  
    lis_la_valeur_des_pixels(i, FICHIER_TXT_ROSSIGNOL1)  
else:  
    lis_la_valeur_des_pixels(i, FICHIER_TXT_ROSSIGNOL2)
```

Cette première capture d'écran permet de lire les valeurs de tous les pixels des deux images automatiquement. En premier lieu, elle les ouvre puis les envoie dans la fonction ci-dessous qui permet de lire les pixels.


```
def lis_la_valeur_des_pixels(i, nom_du_fichier: str):  
    print(nom_du_fichier)  
    (l, h) = i.size  
    liste_values_pixels = []  
    for y in range(h):  
        for x in range(l):  
            pixel = Image.getpixel(i, (x, y))  
            liste_values_pixels.append(pixel)  
  
    if nom_du_fichier == FICHIER_TXT_ROSSIGNOL1:  
        if os.path.exists(IMAGE_ROSSIGNOL1_STORED):  
            efface_un_fichier(IMAGE_ROSSIGNOL1_STORED)  
            ecris_dans_un_fichier(nom_du_fichier, liste_values_pixels)  
        else:  
            ecris_dans_un_fichier(nom_du_fichier, liste_values_pixels)  
  
    if nom_du_fichier == FICHIER_TXT_ROSSIGNOL2:  
        if os.path.exists(IMAGE_ROSSIGNOL2_STORED):  
            efface_un_fichier(IMAGE_ROSSIGNOL2_STORED)  
            ecris_dans_un_fichier(nom_du_fichier, liste_values_pixels)  
        else:  
            ecris_dans_un_fichier(nom_du_fichier, liste_values_pixels)
```

Ce programme récupère l'image que l'on a envoyé ci-dessus et parcourt, comme dans une matrice, tout les pixels de l'image. Nous ajoutons tout ces pixels dans une liste que nous écrivons et stockons dans un fichier.

De là, nous avons déjà une bonne avancée sur l'énigme puisque que nous nous sommes retrouvés avec deux images qui sont d'apparences identique mais qui ne le sont pas du tout dans les pixels puisque avec ces deux fichiers .txt qui stockent tous les pixels respectifs des deux images, nous nous retrouvons avec deux fichiers ci-dessus :

```
src >  pixels_rossignol1.txt
1 [216, 216, 216, 217, 218, 217, 216, 214, 216, 220, 220, 217, 216, 222, 225, 224, 221, 223, 225, 224, 221, 220, 222, 224, 213, 222, 225, 217, 208, 210, 213, 216, 220, 225, 227, 220, 214, 224,
227, 216, 221, 217, 213, 215, 219, 219, 213, 207, 215, 216, 212, 214, 221, 218, 216, 223, 219, 217, 215, 215, 218, 220, 222, 222, 223, 225, 222, 217, 213, 217, 221, 223, 222, 219, 213, 210,
210, 215, 220, 222, 213, 220, 222, 217, 218, 226, 230, 227, 222, 216, 215, 219, 222, 220, 215, 212, 211, 215, 217, 216, 215, 216, 221, 224, 212, 213, 220, 229, 229, 222, 219, 223, 214, 218,
223, 224, 221, 217, 221, 228, 228, 224, 221, 220, 223, 224, 222, 218, 225, 221, 217, 216, 218, 220, 220, 220, 221, 216, 217, 221, 222, 217, 216, 220, 217, 218, 220, 220, 220, 219, 219, 219,
217, 217, 210, 222, 225, 224, 220, 217, 222, 217, 214, 217, 219, 218, 218, 218, 229, 222, 216, 219, 222, 218, 212, 208, 216, 216, 220, 222, 219, 212, 209, 211, 216, 221, 223, 223, 223, 224,
223, 221, 220, 213, 215, 218, 213, 211, 212, 209, 214, 219, 224, 224, 221, 218, 217, 216, 220, 220, 221, 221, 222, 221, 221, 219, 217, 216, 219, 221, 221, 216, 212, 213, 209, 205, 207,
217, 226, 226, 220, 222, 221, 220, 220, 221, 223, 225, 227, 218, 212, 213, 220, 223, 219, 218, 222, 217, 217, 218, 220, 223, 225, 226, 225, 230, 227, 223, 220, 219, 222, 225, 227, 218, 222,
223, 221, 223, 226, 220, 209, 213, 223, 226, 218, 213, 217, 218, 215, 220, 218, 217, 217, 215, 213, 217, 223, 226, 221, 218, 219, 219, 220, 227, 235, 226, 222, 219, 218, 220, 220, 218, 216,
220, 222, 221, 219, 219, 221, 220, 216, 223, 227, 228, 222, 216, 214, 217, 220, 223, 220, 216, 216, 219, 223, 226, 228, 214, 219, 225, 226, 222, 218, 215, 215, 218, 217, 216, 216, 222, 226,
226, 221, 219, 213, 209, 215, 224, 228, 222, 214, 209, 212, 219, 227, 227, 222, 219, 220, 229, 223, 222, 223, 220, 215, 218, 226, 222, 221, 220, 218, 219, 222, 226, 228, 221, 219, 218, 221,
224, 224, 219, 214, 221, 224, 226, 224, 218, 213, 210, 209, 227, 226, 222, 218, 216, 216, 217, 218, 220, 225, 225, 220, 221, 225, 223, 215, 222, 221, 221, 221, 223, 225, 227, 228, 232, 228,
224, 224, 226, 228, 227, 224, 222, 218, 219, 220, 217, 222, 223, 213, 216, 215, 216, 218, 221, 223, 223, 223, 223, 222, 223, 224, 223, 219, 221, 225, 226, 221, 218, 220, 221, 218, 218, 221,
212, 204, 208, 223, 224, 214, 216, 228, 227, 222, 216, 215, 217, 218, 216, 213, 220, 221, 219, 216, 213, 213, 216, 219, 223, 224, 226, 228, 224, 220, 223, 228, 223, 221, 218, 216, 215, 216,
218, 219, 216, 216, 216, 220, 223, 224, 220, 217, 230, 224, 219, 219, 223, 224, 224, 223, 223, 218, 222, 226, 218, 215, 221, 223, 203, 210, 216, 218, 222, 225, 221, 214, 214, 214, 216, 220,
224, 224, 219, 215, 213, 216, 223, 222, 217, 223, 227, 218, 210, 219, 217, 222, 221, 217, 226, 222, 220, 223, 223, 222, 223, 225, 220, 213, 215, 217, 220, 222, 219, 215, 216, 221, 220, 220,
215, 210, 209, 213, 216, 216, 222, 223, 224, 222, 219, 219, 221, 224, 228, 222, 219, 222, 228, 230, 224, 217, 214, 218, 218, 215, 216, 222, 225, 224, 227, 222, 220, 222, 222, 217, 215, 216,
222, 224, 223, 218, 216, 221, 227, 231, 213, 219, 223, 222, 221, 222, 219, 213, 224, 214, 207, 219, 220, 209, 209, 206, 213, 212, 210, 217, 215, 205, 199, 205, 209, 209, 202, 195, 191,
188, 184, 177, 173, 173, 168, 159, 162, 167, 162, 159, 153, 148, 146, 142, 136, 135, 139, 127, 126, 124, 121, 120, 115, 110, 107, 102, 95, 84, 75, 75, 76, 71, 63, 49, 46, 41, 36, 36, 38, 33,
25, 25, 27, 30, 29, 26, 25, 28, 30, 29, 29, 28, 28, 29, 29, 29, 30, 30, 32, 35, 36, 35, 34, 29, 27, 26, 26, 28, 30, 31, 31, 32, 32, 33, 34, 36, 39, 42, 44, 45, 51, 57, 59, 59, 60, 63,
67, 73, 78, 86, 94, 101, 109, 116, 121, 125, 130, 136, 138, 137, 138, 141, 146, 145, 145, 148, 150, 145, 137, 135, 138, 136, 136, 135, 132, 127, 119, 114, 109, 94, 89, 79, 72, 68, 65, 62, 61,
53, 58, 61, 58, 60, 60, 77, 81, 83, 92, 102, 110, 120, 132, 140, 140, 150, 147, 163, 173, 166, 171, 186, 188, 210, 208, 217, 226, 224, 212, 212, 232, 251, 241, 232, 234, 237, 231, 231, 213,
```

Qui concerne l'image 1 et

```
src >  pixels_rossignol2.txt
1 [217, 217, 217, 216, 218, 217, 217, 215, 216, 221, 221, 216, 217, 223, 224, 225, 220, 222, 225, 225, 220, 220, 222, 225, 212, 222, 225, 217, 209, 211, 213, 217, 221, 224, 226, 221, 214, 224,
227, 216, 221, 216, 213, 215, 219, 219, 213, 207, 215, 216, 212, 214, 221, 218, 216, 222, 218, 217, 214, 214, 219, 221, 222, 222, 222, 224, 222, 216, 216, 220, 222, 222, 218, 212, 210,
210, 214, 220, 222, 212, 220, 222, 216, 218, 226, 230, 226, 222, 216, 214, 218, 222, 220, 214, 212, 210, 214, 216, 216, 214, 216, 220, 224, 212, 212, 220, 228, 228, 222, 218, 222, 214, 218,
222, 224, 220, 216, 220, 228, 228, 224, 220, 222, 224, 222, 218, 224, 220, 216, 216, 218, 220, 220, 220, 220, 216, 216, 220, 222, 216, 216, 220, 216, 218, 220, 220, 220, 218, 218, 218,
216, 216, 218, 222, 224, 224, 220, 216, 222, 216, 214, 216, 218, 218, 218, 218, 228, 222, 216, 218, 222, 218, 212, 208, 216, 216, 220, 222, 218, 212, 208, 210, 216, 220, 222, 222, 222, 224,
222, 220, 220, 212, 214, 218, 212, 210, 212, 208, 214, 218, 224, 224, 220, 218, 216, 216, 220, 220, 220, 220, 220, 220, 218, 216, 216, 218, 220, 220, 216, 212, 212, 208, 204, 206,
216, 226, 226, 220, 222, 220, 220, 220, 228, 222, 224, 226, 218, 212, 212, 220, 222, 218, 218, 222, 216, 216, 218, 220, 222, 224, 226, 224, 230, 226, 222, 220, 218, 222, 224, 226, 218, 222,
222, 220, 222, 226, 220, 208, 212, 222, 226, 218, 212, 216, 218, 214, 220, 218, 216, 216, 214, 212, 216, 222, 226, 220, 218, 218, 218, 220, 226, 234, 226, 222, 218, 218, 220, 220, 218, 216,
220, 222, 220, 218, 218, 220, 216, 222, 226, 228, 222, 216, 214, 216, 220, 222, 220, 216, 218, 222, 226, 228, 214, 218, 224, 226, 222, 218, 214, 214, 218, 216, 216, 216, 222, 226,
226, 220, 218, 212, 208, 214, 224, 228, 222, 214, 208, 212, 218, 226, 226, 222, 218, 220, 228, 222, 222, 220, 214, 218, 226, 222, 220, 220, 218, 218, 222, 226, 228, 220, 218, 218, 220,
224, 224, 218, 214, 220, 224, 226, 224, 218, 212, 210, 208, 226, 226, 222, 218, 216, 216, 218, 220, 220, 224, 222, 224, 220, 220, 220, 224, 222, 214, 222, 220, 220, 222, 224, 226, 228, 228,
224, 224, 226, 228, 226, 224, 222, 218, 218, 220, 216, 222, 222, 212, 216, 214, 216, 218, 220, 222, 222, 222, 222, 222, 224, 222, 218, 220, 224, 226, 220, 218, 220, 220, 218, 218, 220,
212, 204, 208, 222, 224, 214, 216, 228, 226, 222, 216, 214, 216, 218, 216, 212, 220, 220, 218, 216, 212, 212, 216, 218, 222, 224, 226, 228, 224, 220, 222, 228, 222, 220, 218, 216, 214, 216,
218, 218, 216, 216, 216, 220, 222, 224, 220, 216, 230, 224, 218, 218, 222, 224, 224, 222, 218, 222, 226, 218, 214, 220, 222, 202, 210, 216, 218, 222, 224, 220, 214, 214, 214, 216, 220,
224, 224, 218, 214, 212, 216, 222, 222, 216, 222, 226, 218, 210, 218, 216, 222, 220, 216, 226, 222, 220, 220, 222, 222, 224, 220, 212, 214, 216, 220, 222, 218, 214, 216, 220, 220, 220,
214, 210, 208, 212, 216, 216, 222, 222, 224, 222, 218, 218, 220, 224, 228, 222, 218, 222, 228, 220, 224, 216, 214, 218, 218, 214, 216, 222, 224, 224, 226, 222, 220, 222, 222, 216, 214, 216,
222, 224, 222, 218, 216, 220, 226, 230, 212, 218, 222, 222, 220, 222, 218, 212, 218, 224, 214, 206, 218, 220, 208, 208, 206, 212, 212, 210, 216, 214, 204, 198, 204, 208, 208, 202, 194, 190,
188, 184, 176, 172, 172, 168, 158, 162, 166, 162, 158, 152, 148, 146, 142, 136, 134, 138, 126, 126, 124, 120, 120, 114, 110, 106, 102, 94, 84, 74, 74, 76, 70, 62, 48, 46, 40, 36, 36, 38, 32,
24, 24, 26, 30, 28, 26, 24, 28, 30, 28, 28, 28, 28, 28, 28, 28, 30, 30, 32, 34, 36, 34, 34, 28, 26, 26, 26, 28, 30, 30, 30, 32, 32, 32, 34, 36, 38, 42, 44, 44, 50, 56, 58, 58, 60, 62,
66, 72, 78, 86, 94, 100, 108, 116, 120, 124, 130, 136, 138, 136, 138, 140, 146, 144, 144, 148, 150, 144, 136, 134, 138, 136, 136, 134, 132, 126, 118, 114, 108, 94, 88, 78, 72, 68, 64, 62, 60,
52, 58, 60, 58, 60, 60, 68, 76, 80, 82, 92, 102, 110, 120, 132, 140, 140, 150, 146, 162, 172, 166, 170, 186, 188, 210, 208, 216, 226, 224, 212, 212, 222, 250, 240, 232, 234, 236, 230, 220, 212,
```

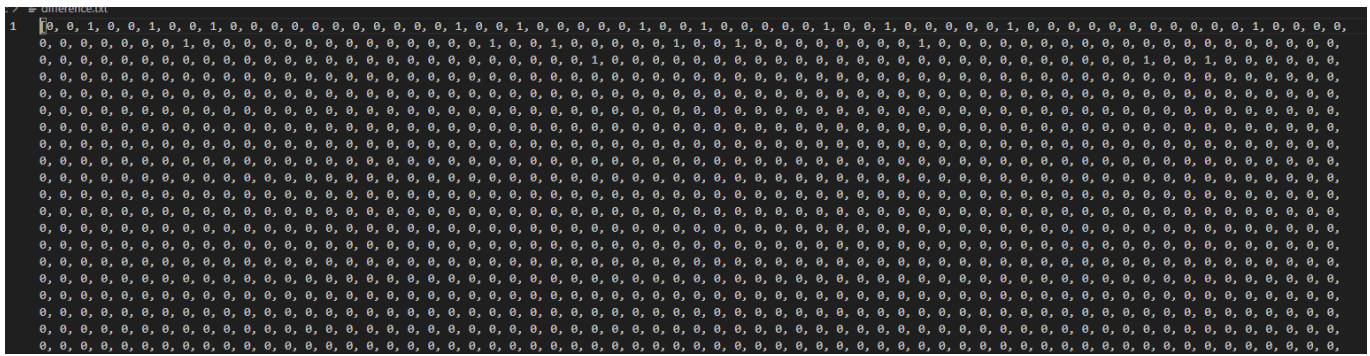
Qui concerne l'image 2.

Et c'est là que notre réflexion s'est avérée être bonne puisqu'au simple regard des premières lignes des deux fichiers en les comparant, nous tombons sur des pixels en décimaux qui sont différents avec par exemple le tout premier pixel de la première capture d'écran qui est à 216 et le tout premier pixel de la seconde capture d'écran qui est à 217.

Le plus déterminant dans tout ça est que quand nous analysons les pixels à premières vues, nous avons vues qu'ils étaient tous différents de 1. Nous avons donc poser le programme suivant qui permet d'écrire et de stocker dans un fichier txt les différences entre chaque pixel des deux images :

```
def compare_deux_images():
    liste_difference = []
    for pixel_image_1, pixel_image_2 in
zip(lit_le_fichier(IMAGE_ROSSIGNOL1_STORED),
lit_le_fichier(IMAGE_ROSSIGNOL2_STORED)):
        if pixel_image_2.isdigit() and pixel_image_1.isdigit():
            liste_difference.append((int(pixel_image_2) - int(pixel_image_1)))

    if os.path.exists(DIFFERENCE_FILE_IMAGE_1_AND_IMAGE_2_STORED):
        efface_un_fichier(DIFFERENCE_FILE_IMAGE_1_AND_IMAGE_2_STORED)
        ecris_dans_un_fichier(DIFFERENCE_FILE_IMAGE_1_AND_IMAGE_2,
liste_difference)
    else:
        ecris_dans_un_fichier(DIFFERENCE_FILE_IMAGE_1_AND_IMAGE_2,
liste_difference)
    return liste_difference
```

Les 1 ne se forment qu'au début, tout le reste du fichier sont des bits à 0. Nous venons de découvrir la clé qui est caché dans les deux images.

Ici, pour le rendu de cette clé et tout le processus derrière, c'est ce qu'on nous appelons de la stéganographie. Un procédé permettant de dissimuler un message au sein des données. Ce procédé est complètement celui utilisé ici et permet de pouvoir communiquer sans que personne ne se doute de rien. Là est l'avantage de ce procédé.

Ainsi, on extrait le clé suivante de 64 bits :

```
[0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

Partie 3

Contexte :

"Suivez les conseils de Mr Robot et proposez un programme Python, utilisant scapy, qui vous permette de filtrer les messages reçus afin de ne conserver que ceux d'Alice et Bob, puis déchiffrer les données transportées par ces paquets."

Ici, nous avons à notre disponibilité un README.md et un fichier .cap.

Pour nous aider dans les démarches d'analyse, il nous est conseillé d'utiliser scapy. Ainsi, commençons notre analyse. Grâce aux détails du README.md nous savons que *Cette trace contient des messages correspondant à des échanges entre Alice et Bob utilisant le protocole **plutotBonneConfidentialité**. La partie asymétrique du protocole (échange de la clé de session) a déjà eu lieu. Ils utilisent maintenant cette clé afin de chiffrer symétriquement leur communication grâce à l'algorithme **AES-256**. Vous avez normalement récupéré une partie de la clé, de taille 64 bits, à l'étape précédente. Il vous suffira de répliquer **4 fois** cette clé afin d'obtenir la clé utilisée par Alice et Bob.*

Ici, pour commencer notre analyse, nous savons que nous réutilisons la suite binaire récupérée dans la **Partie 2** et qui est notre clé de session. On nous dit par ailleurs qu'il faut la dupliquer 4 fois en sachant que nous utilisons l'algorithme **AES-256**. Ainsi, on pose le programme suivant qui duplique la clé :

```
def duplique_cle_de_session(cle):
    cle*=4
```



```
return cle
```

Pour la suite, nous devons, comme indiqué dans le README.md, analyser via un programme python la trace réseau. Pour cela nous allons utiliser la librairie scapy. De plus, nous savons que :

- Le protocole **plutotBonneConfidentialite** est un protocole applicatif et utilise UDP sur le port 9999.
- Avant d'être envoyé, on ajoute en tête du message le vecteur d'initialisation (16 octets) utilisé par **AES**.
- Le protocole AES est utilisé en mode CBC. Cela implique que la taille des messages envoyés doit être un multiple de la taille des blocs, soit un multiple de \$128\$ bits. Pour cela, un mécanisme de *remplissage* (padding) est utilisé. Il s'agit de **PKCS7** (présent dans la bibliothèque python **cryptography** par exemple).

A partir de là, déjà nous savons que nous allons utiliser la librairie scapy et diriger notre programme sur le protocole UDP et le port 9999.

Ainsi, on pose le programme suivant :

```
def analyser_fichier_cap(fichier_cap):  
    packets = rdpcap(fichier_cap)  
    udp_packets = [packet for packet in packets if "UDP" in packet and  
        packet["UDP"].dport == 9999]  
    for packet in udp_packets:  
        print(packet.summary())
```

Le programme ci-dessus lit en premier lieu le fichier .cap avec la fonction **rdpcap(fichier_cap)** puis récolte tous les paquets du fichier .cap qui sont inclut dans le protocole UDP et qui sont sur le port 9999. Ainsi, de par cette fonction, nous filtrons que les informations dont nous avons besoin.

On arrive finalement au résultat suivant en utilisant la trace réseau donnée :

```
Ether / IP / UDP 10.0.0.5:9999 > 10.0.0.6:9999 / Raw  
Ether / IP / UDP 10.0.0.6:9999 > 10.0.0.5:9999 / Raw
```

Ici, pour décrire brièvement ce que la fonction nous renvoie, il est important de comprendre le principe d'encapsulation des données. En premier lieu nous avons la protocole Ethernet qui enveloppe le protocole IP qui lui-même enveloppe le protocole UDP (sur l'adresse IP 10.0.0.5 et sur le port 9999) et enfin qui enveloppe le contenu du paquet 'RAW'. De plus, on remarque un envoi et un reçu. En premier lieu, un envoi de 10.0.0.5:9999 vers 10.0.0.6:9999 puis un autre envoi, semblable à une réponse à première vue, de 10.0.0.6:9999 vers 10.0.0.5:9999.

Ainsi, nous avons filtrer notre trace mais ce que nous voulons est dans la partie RAW puisqu'elle contient le contenu du paquet qui nous intéresse. Ajoutons donc quelques lignes à notre programme qui nous permettra de nous retourner le contenu de ces paquets :

```
def analyser_fichier_cap(fichier_cap):
    packets = rdpcap(fichier_cap)
    udp_packets = [packet for packet in packets if "UDP" in packet and
packet["UDP"].dport == 9999]
    for packet in udp_packets:
        print(packet.summary())
        print("Contenu du paquet :")
        if Raw in packet:
            print(packet[Raw].load)
```

La fonction ci-dessus nous retourne les valeurs suivantes :

```
b'\xe5\xc4\x17\x1c\xb3\x030\x99\xb6I\xcc\xf847\x8c\x9b\xd7\xc4\x87\x16U\x84gh]\xe7
\xd8\xc8\x8e\x0c\xcf\xb7\xf7\xe2hy\x1b\x181z\x9d\xae\x05\xa7'\xfe\xed'
```

Pour le premier envoi et :

```
b'\xd5\x8a\xc5\xbd,:D\x96\xc0\xcbz\x80@G\xec\x83\x07%S\x82V\x9d\xd2\x9d\x81\xf3\xb
7(\x17\xeb%\x9a\xdcG\xca\xden=S\x97WWW\\ \x98\x95mR'
```

Pour le deuxième envoi. Notons que les contenus des paquets que nous obtenons sont de l'héxadécimal (ce qui nous rappelle le chiffage AES sur lequel nous avons travailler dans la partie 2, nous sommes donc sûr que nous tenons la bonne piste).

Ainsi, nous avons la clé de session, le chiffage utilisée et les modules pour déchiffrer les messages avec les fonctions pour AES établies lors de la partie 2. Il nous manque plus qu'une fonction qui nous permettra de déchiffrer tout cela. Ainsi, on pose le programme suivant qui permet de déchiffrer le contenu d'un paquet selon la valeur recueillie (en héxadécimal) et la clé de session.

Avant cela, nous devons convertir la clé de session que nous avons obtenus (je le rappelle en binaire) pour en faire une clé de session en valeur décimal. Notre fonction de déchiffage selon AES utilise une clé de session en décimal.

Ainsi, on pose le programme suivant qui convertit notre liste de valeur binaire en valeur décimal correspondante :

```
def binaire_to_decimal(cle_de_session: list[int]):
    binaire_en_string = ''.join(str(bit) for bit in cle_de_session)
    valeur_decimal_correspondante = int(binaire_en_string, 2)
    return valeur_decimal_correspondante

print("\nClé de session dupliqué en décimal\n" +
str(binaire_to_decimal(cle_duplique)))
```

qui nous retourne pour notre clé de session (multipliée par 4 en début d'analyse) la valeur suivante :

```
16509481210563468426600490343268981929809196587515063843521106276554784178192
```

Durant notre avancement, nous avons oublié quelque chose. On note que le vecteur d'initialisation est dans les 16 premiers octets du paquet, ainsi nous savons que les 16 premiers octets sont pour l'IV et le reste pour le contenu du message. On modifie donc encore un peu plus notre programme pour récupérer les vecteurs d'initialisations de chaque paquet respectif. On donne l'évolution programme suivant :

```
def analyser_fichier_cap(fichier_cap):
    packets = rdpcap(fichier_cap)
    udp_packets = [packet for packet in packets if "UDP" in packet and
        packet["UDP"].dport == 9999]
    contenu_des_paquets = []
    liste_IV = []
    for packet in udp_packets:
        print(packet.summary())
        print("\nContenu du paquet :")
        if Raw in packet:
            contenu_des_paquets.append(packet[Raw].load[16:])
            print(packet[Raw].load[16:])
        iv = packet.load[:16]
        liste_IV.append(iv)
        print("IV : " + str(iv) + "\n")
    return contenu_des_paquets, liste_IV
```

Et qui nous renvoie les données suivantes :

```
Ether / IP / UDP 10.0.0.5:9999 > 10.0.0.6:9999 / Raw
```

Contenu du paquet :

```
b"\xd7\xc4\x87\x16U\x84gh]\xe7\xd8\xc8\xe8\x0c\xcf\xb7\xf7\xe2hy\x1b\x181z\x9d\xae
\x05\xa7'\!\xfe\xed"
```

```
IV : b'\xe5\xc4\x17\x1c\xb3\x030\x99\xb6I\xcc\xf847\x8c\x9b'
```

```
Ether / IP / UDP 10.0.0.6:9999 > 10.0.0.5:9999 / Raw
```

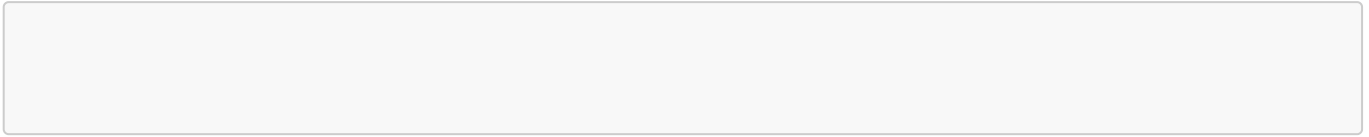
Contenu du paquet :

```
b'\x07%S\x82V\x9d\xd2\x9d\x81\xf3\xb7(\x17\xeb%\x9a\xdcG\xca\xden=S\x97WWW\\\x98\x
95mR'
```

```
IV : b'\xd5\x8a\xc5\xbd,:D\x96\xc0\xcbz\x80@G\xec\x83'
```

Et enfin, la dernière chose à prendre en compte dans notre déchiffrement est que : *Le protocole AES est utilisé en mode CBC. Cela implique que la taille des messages envoyés doit être un multiple de la taille des blocs, soit un multiple de 128 bits. Pour cela, un mécanisme de remplissage (padding) est utilisé. Il s'agit de PKCS7 (présent dans la bibliothèque python cryptography par exemple).*

Ce qui signifie que nous devons faire du remplissage dans le message envoyé afin de pouvoir le déchiffrer avec notre clé de session. On expose donc le programme qui déchiffre suivant en sachant que nous avons notre clé de session multipliée, nos contenus des paquets, nos IV respectifs et notre module AES. Testons donc la fonction suivante qui permet de retrouver les messages en clair des paquets :



Malheureusement, nous n'avons pas réussi à trouver la solution en termes de développement pur et dur

Partie 4

Alice et Bob utilisent toujours la même clé. Est-ce une bonne pratique ?

Non, ce n'est pas une bonne pratique car si quelqu'un arrive à trouver la clé, il pourra décoder tous les prochains messages envoyés par Alice et Bob. C'est donc une faille de sécurité, pour palier cela, il pourrait par exemple, générer une clé aléatoire toutes les semaines qu'ils s'enverraient de la même manière. Ainsi, même si un hacker se trouve au milieu des échanges d'Alice et Bob, il devra toutes les semaines recraquer la clé échangée

Le protocole PlutotBonneConfidentialité est inspiré d'un vrai protocole réseau. Lequel? Décrivez la partie associé à la certification des clés qui est absente de PlutotBonneConfidentialité.

Le protocole "PlutotBonneConfidentialité" est inspiré du protocole TLS et SSL. La partie qui est absente est la certification des clés. En effet, dans le protocole TLS et SSL, il y a un tiers de confiance qui certifie les clés, ce qui permet de certifier que la clé est bien celle de la personne avec qui on communique. Cela permet aussi de certifier que l'on communique bien avec le serveur que l'on souhaite.

Il n'y a pas que pour l'échange de mots doux q'un tel protocole peut se révéler utile. . . Donnez au moins deux autres exemples de contexte où cela peut se révéler utile

Le SSL est utilisé dans les transmissions entre un site web et un navigateur pour sécuriser les échanges. On peut aussi le retrouver dans les transaction bancaire pour sécuriser les données échangées. La force de ce protocole est donc approuvée.

Connaissez-vous des applications de messagerie utilisant des mécanismes de chiffrement similaires? (on parle parfois de chiffrement de bout en bout)? Citez-en au moins deux et décrivez brièvement les mécanismes cryptographiques sous-jacen

WhatsApp utilise le chiffrement de bout en bout pour sécuriser les messages entre utilisateurs. Il s'appuie sur le protocole Signal, développé par la Signal Foundation. Ce protocole utilise des clés publiques et privées pour chaque utilisateur. Lorsqu'un message est envoyé, il est chiffré localement sur le téléphone de l'émetteur à l'aide de la clé publique du destinataire. Le message chiffré est ensuite envoyé au destinataire et seul ce dernier peut le déchiffrer à l'aide de sa clé privée.

On peu aussi parler de telegram qui permet aux utilisateurs de choisir pour leurs conversations de les crypter avec un protocole de cryptage de bout en bout appelée « Chats Secrets ». Lorsqu'un utilisateur active ce mode pour une conversation spécifique, elle est chiffrée de bout en bout. Le mécanisme précis de chiffrement

utilisé dans Telegram n'est pas entièrement ouvert au public et il y a eu des débats sur sa sécurité et sa fiabilité par rapport à d'autres solutions de chiffrement. Cependant, Telegram affirme utiliser des algorithmes de chiffrement forts pour sécuriser ces conversations.

Récemment, différents projets de loi et règlements (CSAR, EARN IT Act) visent à inciter voir obliger les fournisseurs de services numériques à pouvoir déchiffrer (et donc analyser) les communications de leur.e.s utilisateur.rices. Discutez des arguments en faveur ou contre ces législations, notamment en matière de vie privée

Pour les arguments contre ces lois on pourrait parler du respect de la vie privée des utilisateurs qui ne souhaitent pas forcément voir leurs messages privés être visible par autrui et de ce qu'ils concernent. Cependant pour les arguments pour ses lois, on pourrait citer le but de la loi CSAR qui vise les fournisseurs de contenu à détecter des contenus d'abus sexuels de mineurs en analysant les conversations de leurs utilisateurs. Nous savons que les réseaux sociaux sont les premiers points de contacts de beaucoup d'abus sexuels, de pédophilie, de pédopornographie ou tout autres sujets aussi grave que ceux-là. L'argument en faveur de ces lois serait donc que cela permettrait de diminuer ce genre de contenu ou de diminuer de potentiels actes malveillants. Nous savons qu'aujourd'hui, les enfants sont de plus en plus tôt connectés sur les réseaux mais que leur naïveté, elle, est beaucoup trop grande pour pouvoir les faire prendre en compte les dangers d'internet. C'est donc pour cela que ces lois pourraient être utiles.

Répartition des tâches

Pour la répartition des tâches, vous pouvez consulter le git mais la réflexion n'est pas visible. Ainsi, nous précisons cela car pour la partie 1, nous étions tous les deux inclus.

Plus précisément, nous avons réfléchi tous les deux sur les réponses mais pour le développement, Gael a fait le passage astucieux et Steven le passage brutal.

La partie 2 a été majoritairement faite par les deux membres du binôme mais plus précisément, nous répartissons les tâches suivantes :

- question 2.1 : Gael
- question 2.2 : Gael
- question 3 : Steven
- question 4 (images) : Steven

Pour la partie 3, l'entièreté des réponses ont été faite par Steven.

Et enfin, la partie 4 a été entièrement faite par Gael.