SAÉ Crypto - Partie 2

BOURREAU Quentin
CHHUM–MOXEL Loann

BUT Informatique 2.3.B
2023-2024

Sommaire

Partie 1: premières tentatives

- En supposant que RSA soit utilisé correctement, Eve peut-elle espérer en venir à bout?

RSA est un algorithme de cryptographie asymétrique très utilisé. Il repose sur la difficulté de factoriser de grands nombres premiers. Si RSA est correctement mis en œuvre avec une taille de clé adéquate, il offre une sécurité considérable. Par conséquent, tant que RSA est utilisé correctement et que la clé de session générée est suffisamment robuste, Eve ne devrait pas être en mesure de déchiffrer la communication ou à moins de tenter de déchiffrer la communication depuis extrêmement longtemps.

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

L'algorithme SDES (Simplified DES) est peu sécurisé en raison de sa faible longueur de clé. SDES est un algorithme cryptographique développé par Edward Schaefer en 1986 à des fins éducatives. Effectivement, le DES original (Data Encryption Standard) utilisait une clé de 56 bits, mais la version simplifiée SDES réduit cette longueur à seulement 10 bits (1 bit de parité inclus). Ainsi, cela rend l'algorithme très vulnérable aux attaques de force brute.

Tout d'abord, la force brute consiste à essayer toutes les clés possibles jusqu'à ce que la bonne clé soit trouvée. Avec uniquement 10 bits de clé, le nombre total de clés possibles est de 2**10, soit 1024 clés. En comparaison à une attaque de force brute sur DES, celle-ci nécessiterait 2**56 essais, ce qui est extrêmement difficile à réaliser même avec la puissance de calcul actuelle.

Par conséquent, en analysant le nombre d'essais nécessaire à une méthode de force brute, on peut conclure que l'algorithme SDES est peu sécurisé. En utilisant une approche exhaustive, Eve pourrait rapidement essayer toutes les clés possibles et déchiffrer le message en utilisant seulement 1024 tentatives, ce qui est faisable

même sans des ressources informatiques très avancées. De plus, en moyenne, les messages cryptés avec le SDES peuvent même être cassés par force brute après seulement 512 essais.

- Est-ce que double SDES est-il vraiment plus sûr ? 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 ?

Double SDES est une amélioration par rapport à SDES, mais il n'est pas suffisamment sécurisé pour résister à certaines attaques, notamment l'attaque du milieu. Dans Double SDES, le message est chiffré deux fois avec deux clés différentes, comme indiqué par l'équation suivante :

$$C = SDES(SDES(M, k1), k2)$$

Par conséquent, l'attaque du milieu sur Double SDES est basée sur le fait qu'un attaquant dispose de paires de messages clairs et de leurs correspondants chiffrés. Ainsi, dans notre cas avec Eve, on suppose qu'elle connaîtrait M (message clair) et C (message chiffré). Le processus d'attaque du milieu se compose en deux étapes :

- Attaque par force brute sur la première étape de chiffrement (utilisant k1):
    - Eve chiffre le message clair M avec toutes les 2**10 clés possibles pour k1.
    - Elle stocke les paires (C1, k1), où C1 est le résultat chiffré
- Attaque par force brute sur la deuxième étape de chiffrement (utilisant k2):
    - Eve déchiffre le message chiffré C avec toutes les 2**10 clés possibles pour k2.
    - Eve cherche pour chaque clé k2, s'il y a une correspondance entre les paires (C1, k1) et (M′, k2), où M' est le message déchiffré avec k2.

- Si elle trouve une correspondance, elle aura donc trouvé les clés k1 et k2 correctes utilisées.

Par conséquent, le nombre total d'essais nécessaires le temps de cassage de l'algorithme brutal est le produit des deux étapes, soit 2**10 × 2**10 = 1 048 576 essais. Même si cela paraît élevé, cela reste relativement faisable en comparaison avec la force brute sur une seule couche de SDES, double SDES n'est donc pas réellement fiable car on peut le casser assez rapidement. Néanmoins, double SDES est quand même plus sécurisé que la simple utilisation de SDES.

De plus, au niveau des fonctions de cassage brutal et astucieux, nous avons remarqué que nous pouvions trouver plusieurs couples de clés possibles à partir d'un message clair et un message chiffré.

Pour finir, nous avons réalisé une expérience sur nos fonctions de cassage brutal et astucieux pour mieux voir la différence entre celles-ci. Tout d'abord, nous avons ajouté un argument dans chacune de nos deux fonctions pour savoir si on voulait afficher le nombre de tentatives avant de trouver le bon couple de clés. Ainsi, nous avons réalisé ces deux fonctions de test suivantes :

```python
def test_cassage_brutal():
    """
    Test le cassage brutal et affiche le temps d'exécution du cassage brutal
    sur le texte dans les constantes
    """
    start_time = time.time()
    texte_chiffre = double_chiffrement_sdes(CONST.PREMIERE_CLE_TEST,
                                            CONST.DEUXIEME_CLE_TEST, CONST.TEXTE_TEST)
    premiere_cle_trouvee, deuxieme_cle_trouvee = cassage_brutal(CONST.TEXTE_TEST,
                                                                texte_chiffre, True)
    if premiere_cle_trouvee is not None and deuxieme_cle_trouvee is not None:
        print("Clé 1 :", bin(premiere_cle_trouvee))
        print("Clé 2 :", bin(deuxieme_cle_trouvee))
        texte_dechiffre_1 = dechiffrement_sdes(int(bin(premiere_cle_trouvee), 2), texte_chiffre)
        texte_dechiffre_2 = dechiffrement_sdes(int(bin(deuxieme_cle_trouvee), 2), texte_dechiffre_1)
        print("Message original :", texte_dechiffre_2)
    else:
        print("Aucune clé trouvée")
    end_time = time.time()
    execution_time = end_time - start_time
    print("Temps d'exécution cassage brutal :", execution_time, "secondes\n")
```

```python
def test_cassage_astucieux():
    """
    Test le cassage astucieux et affiche le temps d'exécution du cassage astucieux sur le
    texte dans les constantes
    """
    start_time = time.time()
    texte_chiffre = double_chiffrement_sdes(CONST.PREMIERE_CLE_TEST,
                                            CONST.DEUXIEME_CLE_TEST, CONST.TEXTE_TEST)
    premiere_cle_trouvee, deuxieme_cle_trouvee = cassage_astucieux(CONST.TEXTE_TEST,
                                                                   texte_chiffre, True)
    if premiere_cle_trouvee is not None and deuxieme_cle_trouvee is not None:
        print("Clé 1 :", bin(premiere_cle_trouvee))
        print("Clé 2 :", bin(deuxieme_cle_trouvee))
        texte_dechiffre_1 = dechiffrement_sdes(int(bin(premiere_cle_trouvee), 2), texte_chiffre)
        texte_dechiffre_2 = dechiffrement_sdes(int(bin(deuxieme_cle_trouvee), 2), texte_dechiffre_1)
        print("Message original :", texte_dechiffre_2)
    else:
        print("Aucune clé trouvée")
    end_time = time.time()
    execution_time = end_time - start_time
    print("Temps d'exécution cassage astucieux :",execution_time,"secondes\n")
```

Suite au lancement de ces dernières, nous obtenons un affichage comme ceci :

```
Nombre de tentatives pour trouver le couple de clés : 2472
Clé 1 : 0b110100111
Clé 2 : 0b10
Message original : test
Temps d'exécution cassage brutal : 0.8667447566986084 secondes

Nombre de tentatives pour trouver le couple de clés : 1448
Clé 1 : 0b110100111
Clé 2 : 0b10
Message original : test
Temps d'exécution cassage astucieux : 0.3137495517730713 secondes
```

Pour conclure, pour le nombre de tentatives, on voit bien que le cassage brutal prend plus de temps que le cassage astucieux et que le nombre de tentatives est plus élevé. Bien évidemment, ici l'écart du nombre de tentatives et le temps d'exécution entre les deux ne semble pas si grand, cela s'explique notamment car nous avons testé ceci sur le texte "test" qui n'est pas long. Néanmoins, si nous testons sur un des textes données dans le sujet, la nous voyons complètement la différence entre le cassage brutal et le cassage astucieux qui est beaucoup plus rapide et demande moins de tentatives. Nous avons essayé de tester sur un des fichiers donnés dans le sujet, mais le cassage brutal prenait trop de temps donc nous avons stoppé ce test.

Puis, pour finir cette expérience, nous avons également testé la moyenne du temps d'exécution de cassage brutal et astucieux en ajoutant des tests dans nos programmes. Effectivement, nous avons pris le même message clair et message chiffré et avons effectué une boucle de cent fois chaque cassage pour calculer le temps d'exécution moyen de chaque cassage..Nous obtenons ainsi l'affichage suivant et qui nous montre encore que le cassage astucieux est plus efficace que le cassage brutal :

```
Voici le texte utilisé lors du cassage brutal: test
La moyenne du temps d'exécution du cassage brutal de double SDES est de : 0.5885664439201355 secondes
Voici le texte utilisé lors du cassage astucieux: test
La moyenne du temps d'exécution du cassage astucieux de double SDES est de : 0.1600193500518799 secondes
```

Partie 2: Un peu d'aide

- Est-ce vraiment un problème?

L'utilisation de l'algorithme AES avec une clé de taille 256 bits crée un vrai problème. En effet, l'utilisation d'un chiffrement avec une clé aussi grande ne permet plus l'utilisation de méthodes de cassages dites brutales. Pour voir cela, il suffit de comparer le nombre d'essais nécessaires pour chaque technique de chiffrement. Pour SDES : 2**10 soit 1024 essais nécessaires, et pour AES, 2**256 soit 1,15 * 10 ** 77 ce qui est beaucoup plus élevé. On peut donc se rendre compte que tester chaque possibilité de clé prendrait beaucoup trop de temps et est donc impossible avec les moyens informatiques actuels.

- Illustrer expérimentalement les différences entre les deux protocoles

1) Analyse du temps d'exécution du chiffrement/déchiffrement d'un message

- AES

Le AES (Advanced Encryption Standard) est un algorithme de chiffrement symétrique largement utilisé et reconnu pour la sécurité des données. Le AES utilise une clé symétrique pour chiffrer et déchiffrer les données. Les tailles de clé couramment utilisées sont 128 bits, 192 bits et 256 bits (ici nous devions utiliser une clé de 256 bits). Ces tailles de clé déterminent le niveau de sécurité offert par l'algorithme, avec des clés plus longues généralement considérées comme plus sécurisées.

Pour tester le temps d'exécution de chiffrement et déchiffrement pour AES et SDES, nous avons testé chacune de nos fonctions sur le fichier txt donné dans le sujet (arsene_lupin_extrait.txt). Voici premièrement, notre fonction de chiffrement AES :

```
def chiffrement_aes(cle, message_clair):
    """
    Chiffre un texte de taille quelconque avec AES

    Args:
        cle (int): la clé de chiffrement
        message_clair (str): le message en clair à chiffrer

    Returns:
        (str): le message chiffré avec la clé de chiffrement
    """
    donnees_bytes = cle.to_bytes(32, 'big')
    vecteur_initialisation = b'\x00' * 16
    cipher = Cipher(algorithms.AES(donnees_bytes), modes.CFB(vecteur_initialisation),
                    backend=default_backend())
    encryptor = cipher.encryptor()
    message_clair_bytes = message_clair.encode('utf-8')
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    message_clair_padded = padder.update(message_clair_bytes) + padder.finalize()
    message_chiffre = encryptor.update(message_clair_padded) + encryptor.finalize()
    return b64encode(message_chiffre)
```

Ainsi, nous avons réalisé une boucle de 1000 appels pour tester en moyenne le temps d'exécution du chiffrement. Cela nous a donné :

5.888938903808594e-05 secondes

Ensuite, pour la fonction de déchiffrement de AES (sans les commentaires et docstrings), nous avons celle ci-dessous :

```python
def dechiffrement_aes(cle, message_chiffre):
    """
    Déchiffre un texte de taille quelconque avec AES

    Args:
        cle (int): la clé de chiffrement
        message_clair (str): le message en clair à déchiffrer

    Returns:
        (str): le message chiffré avec la clé de chiffrement
    """
    donnees_bytes = cle.to_bytes(32, 'big')
    message_chiffre = b64decode(message_chiffre)
    vecteur_initialisation = b'\x00' * 16
    cipher = Cipher(algorithms.AES(donnees_bytes), modes.CFB(vecteur_initialisation),
                    backend=default_backend())
    decryptor = cipher.decryptor()
    message_dechiffre_padded = decryptor.update(message_chiffre) + decryptor.finalize()
    unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
    message_dechiffre = unpadder.update(message_dechiffre_padded) + unpadder.finalize()
    return message_dechiffre.decode('utf-8')
```

Par conséquent, nous avons réalisé également une boucle de 1000 appels pour tester en moyenne le temps d'exécution du déchiffrement. Cela nous a donné :

8.857798576354981e-05 secondes

- SDES

SDES, utilisé depuis le début de la SAÉ, pour la fonction de chiffrement nous avons la fonction (sans les commentaires et docstrings) suivante :

```python
def chiffrement_sdes(cle, message_clair):
    """
    Chiffre un texte de taille quelconque avec SDES

    Args:
        cle (int): la clé de chiffrement
        message_clair (str): le message en clair à chiffrer

    Returns:
        (str): le message chiffré avec la clé de chiffrement
    """
    message_chiffre = ""
    for lettre in message_clair:
        lettre_chiffree = encrypt(cle, ord(lettre))
        message_chiffre += chr(lettre_chiffree)
    return message_chiffre
```

Ainsi, lors de l'appel 1000 de cette fonction sur le fichier txt, cela nous donne une moyenne de temps d'exécution de :

0.030735268115997316 secondes

Puis, pour la fonction de déchiffrement ci-dessous :

```python
def dechiffrement_sdes(cle, message_chiffre):
    """
    Déchiffre un texte de taille quelconque avec SDES

    Args:
        cle (int): la clé de chiffrement
        message_clair (str): le message en clair à déchiffrer

    Returns:
        (str): le message chiffré avec la clé de chiffrement
    """
    message_dechiffre = ""
    for lettre in message_chiffre:
        lettre_dechiffree = decrypt(cle, ord(lettre))
        message_dechiffre += chr(lettre_dechiffree)
    return message_dechiffre
```

Par conséquent, lors de l'appel toujours d'une boucle de 1000 fois, en moyenne elle a un temps d'exécution de :

0.02944176769256592 secondes

Voici un tableau récapitulatif de la moyenne du temps d'exécution (en secondes) sur 1000 appels de chacune de nos fonctions :

Chiffrement AES	Déchiffrement AES	Chiffrement SDES	Déchiffrement SDES
5.888938903808594e-05	8.857798576354981e-05	0.030735268115997316	0.02944176769256592

Pour conclure, pour l'analyse du temps d'exécution de chiffrement/déchiffrement entre les deux protocoles, on voit que nos fonctions AES de chiffrement et déchiffrement sont un peu plus rapides que celles pour SDES.

2) Analyse du temps de cassage d'AES

Étant donné que l'AES est actuellement considéré comme un algorithme de chiffrement robuste, les attaques directes contre AES (comme le cassage brutal ou astucieux) sont actuellement impraticables en raison de la taille des clés utilisées (128, 192 ou 256 bits). Ainsi, la suite de cette question n'est que des estimations du temps nécessaire au cassage d'AES. La formule pour estimer le temps nécessaire en secondes serait donc la suivante :

Temps estimé (en secondes) = Nombre total de clés possibles / Taux de tests par seconde

Pour commencer, nous connaissons déjà le nombre total de clés possibles, car cela dépend du nombre de bits de la clé. Ainsi, comme dans cette question nous avons choisi une clé de 256 bits, il y aura 2**256 clés possibles (soit $1.1579209*10^{77}$).

Ensuite, pour le taux de tests par seconde d'une machine, cela se calcule notamment selon la puissance du processeur de l'ordinateur qui sera chargé de casser le chiffrement. En effet, la capacité du processeur a faire un nombre n de calculs par seconde est variable. Par exemple, sur les ordinateurs de l'IUT, les processeurs sont capables d'effectuer 3,4 milliards de calculs par seconde car ils tournent à 3,4Ghz. En supposant que nous soyons capable d'utiliser le processeur au maximum de ses capacité, et ce, de manière constante, nous pouvons obtenir l'estimation suivante :

$$1.1579209*10^{77} / 3{,}4 * 10^{9} = 3.108108 * 10^{85} \text{ secondes}$$

Cela converti en années reviendrait donc à : 9.8567×10^{77} années.

Nous pouvons donc conclure qu'avec le matériel informatique actuel, bien que plutôt performant, il nous est encore impossible de casser le chiffrement AES dans un intervalle de temps raisonnable. Cela prouve encore plus la robustesse de cette technique de chiffrement.

- Quels sont les autres types d'attaques que de tester les différentes possibilités de clés ?

Il existe plusieurs types d'attaques visant à déterminer les possibilités de clé autre que l'attaque du milieu (met-in-the-middle) présenté précédemment. Ce qu'il faut comprendre, c'est que chaque technique vise à déterminer un type très particulier de faiblesse dans le chiffrement qui est utilisé. Cela peut varier en fonction de la technique de chiffrement, mais aussi du contenu des messages ou encore même d'éléments du contexte du chiffrement. Ainsi, nous pouvons citer plusieurs autres types d'attaques.

Premièrement, nous avons l’attaque par analyse fréquentielle. Cette technique s’applique sur des messages textuels. Effectivement, celle-ci utilise le fait que certaines lettres apparaissent plus ou moins que d'autres, et ce, selon la langue. Il faut donc réussir à associer une fréquence d’apparition d’un caractère dans le message chiffré en le comparant aux fréquences connues dans les différentes langues. Par exemple, dans la langue française, la lettre “e” apparaît beaucoup plus souvent que les autres.

Deuxièmement, nous pouvons citer les attaques qui se basent sur le contexte du chiffrement plutôt que sur la technique utilisée. Par exemple, on connaît le contexte du message, l’envoie d’informations précises sur la consommation d’une machine par exemple. Il suffit de déterminer les valeurs associées à chaque message en interceptant le plus possible. Pour cela, il faut avoir un nombre de valeurs relativement faible. Sinon il devient vite impossible de différencier toutes les valeurs surtout s'il n’y a pas d’intervalle précis.

Enfin, nous pouvons citer l’attaque par clé apparente. Cette technique permet à l’attaquant qui connaît la technique mathématique utilisée de comprendre le mécanisme de chiffrement utilisé, et ce, sans même connaître la valeur de la clé utilisée. Il faut que la clé soit utilisée plusieurs fois et avoir une bonne compréhension des techniques de chiffrement. Cette technique s'applique surtout sur les techniques plutôt simples en cryptographie.

- Analyse des images

Étant donné l’aide de MrRobot dans le sujet, du fait qu’il nous fournit deux images d’apparence identique. Ainsi, nous nous doutions qu’il fallait chercher les différences entre les deux images pour trouver la clé utilisée par Alice et Bob lors de leurs communications, notamment suite au mail d’aide de notre professeur Monsieur GILLET. De plus, dans ce dernier, nous savions maintenant que la deuxième image contenait la clé cherchée et que cette clé était en fait une clé binaire de 64 bits.

Par conséquent, nous avons donc commencé à parcourir les deux images simultanément pixel par pixel en largeur puis en hauteur. Par la suite, pour chaque pixel différent entre les deux images, nous regardions juste le pixel, car nous ne comprenions pas ce qu'il fallait faire. Puis, après avoir discuté de notre blocage avec notre professeur Monsieur GILLET, nous avons eu l'information qu'il fallait uniquement garder, pour le pixel de l'image contenant la clé, le dernier bit en binaire du pixel. Une fois cette étape passée, nous avons concaténé chaque bit pour créer une clé, néanmoins cela nous donnait une clé de taille supérieure à 64 et donc nous savions que nous n'avions pas le bon résultat. Voici notre premier programme pour trouver la clé (qui ne fonctionnait pas) :

```
def get_dernier_bit(binaire):
    return int(binaire, 2) & 1

def trouver_cle_image(image_de_base, image_contenant_cle):
    image_de_base = Image.open(image_de_base)
    image_contenant_cle = Image.open(image_contenant_cle)
    cle_de_session_cachee = ""
    largeur, hauteur = image_de_base.size
    for y in range(hauteur):
        for x in range(largeur):
            pixel_de_base = image_de_base.getpixel((x, y))
            pixel_contenant_cle = image_contenant_cle.getpixel((x, y))
            if pixel_de_base != pixel_contenant_cle:
                cle_de_session_cachee += str(get_dernier_bit(bin(pixel_contenant_cle)))
    return cle_de_session_cachee
```

Par la suite, nous avons donc décidé de faire une soustraction entre le pixel de la deuxième image et de la première image, nous trouvions ainsi uniquement trois résultats possibles : -1, 0 ou 1. Cela s'expliquait car, après avoir affiché pixel par pixel simultanément des deux images, nous trouvions qu'il n'y avait que 1 d'écart entre les pixels maximum. Par conséquent, nous avons commencé par faire cela pour chaque résultat de soustraction :

1 -> 1

0 ou -1 -> 0

C’est là que nous trouvions une clé, néanmoins celle-ci ne correspondait toujours pas à la bonne clé puisque la clé était encore trop grande par rapport au 64 bits requis. Ainsi, c’est pour ça que nous avons cherché, par la suite, une autre manière de trouver la bonne clé. Après quelques réflexions au sein de notre groupe et d’analyse de chaque pixel entre la première image et notamment la deuxième image, car nous savions qu’elle contenait la clé, nous avons découvert que chaque pixel de la 2e image était soit pair, soit impair. Ainsi, nous avons tenté de mettre, lorsque le pixel est pair, un 0 pour la clé, sinon un 1. C’est à ce moment, que nous avons vu que la clé, même qu’elle soit beaucoup trop grande pour 64 bits, qu’elle ne contenait que des 0 à partir de 65 bits. Comme nous avions donc trouvé une clé de 64 bits, nous savions que nous avions le bon résultat, néanmoins, nous voulions réduire notre fonction de recherche de la clé. Ainsi, nous nous sommes limités uniquement à un parcours des 64 premiers pixels en largeur de la deuxième image et avons employé le modulo de 2 pour chaque pixel parcouru pour avoir directement le bit de la clé à ajouter (0 quand c’est pair et 1 quand c’est impair). Voici donc notre fonction optimisée de recherche de clé ci-dessous :

```python
def trouver_cle_image(image_contenant_cle):
    """
    Trouve la clé cachée dans l'image

    Args:
        image_contenant_cle (str): le chemin de l'image où
        chercher la clé

    Returns:
        (str): la clé trouvée dans l'image
    """
    image_contenant_cle = Image.open(image_contenant_cle)
    cle_de_session_cachee = ""
    for pixel in range(CONST.CleImage.NOMBRE_BITS_CLE_IMAGE.value):
        pixel_contenant_cle = image_contenant_cle.getpixel((pixel, 0))
        cle_de_session_cachee += str(pixel_contenant_cle % 2)
    return cle_de_session_cachee
```

Partie 3: Analyse des messages

Grâce aux conseils de Mr Robot, nous donnant une trace d'échanges réseau contenant notamment certains messages chiffrés grâce au protocole "PlutotBonneConfidentialité", nous avons pu déchiffrer les messages échangés entre Bob et Alice. En effet, notre fichier Python fournit un ensemble de fonctions pour déchiffrer les messages échangés entre Bob et Alice à partir d'une trace réseau, en utilisant l'algorithme de chiffrement AES en mode CBC (Cipher Block Chaining). De plus, notre fichier utilise notamment scapy qui est un puissant framework d'analyse et de manipulation de paquets réseau en Python. En effet, dans le cas de la SAÉ, ce dernier permet de déchiffrer les données transportées par les paquets envoyés lors de l'échange entre Bob et Alice. Pour ce faire, nous savions, avec le README.md donné dans le sujet, que la clé de session trouvée précédemment de 64 bits devait être répétée 4 fois pour former une clé de 256 bits qui était la clé utilisée par Bob et Alice. Par conséquent, comme nous avions la clé, il suffisait uniquement de récupérer les paquets à partir d'une trace réseau avec scapy, ce dont nous avons trouvé après quelques recherches sur le site de scapy, la fonction rdpcap. Ensuite, il fallait juste parcourir chaque paquet et lire ses données. Néanmoins, il fallait garder uniquement les paquets UDP de port 9999 (expliqué dans le sujet) pour n'avoir que les messages échangés entre Bob et Alice. Une fois chaque paquet trouvé, il fallait déchiffrer le message chiffré contenu dans les données du paquet avec la clé de 256 bits à notre disposition. Puis, il ne manquait plus que l'affichage de ces messages pour avoir le résultat. Ainsi, voici un résumé des fonctions présentes dans notre fichier :

- Fonction `dechiffrer_aes_cbc` :
    - Cette fonction prend une clé, un vecteur d'initialisation et un texte chiffré en entrée.
    - Elle déchiffre le texte chiffré en utilisant l'algorithme AES en mode CBC.
    - Elle retourne le texte déchiffré.

- Fonction `extraire_messages` :
    - Cette fonction prend une liste de paquets réseau et une clé de 256 bits en entrée.
    - Elle parcourt les paquets pour extraire ceux qui sont reçus en utilisant le protocole UDP et ayant le bon port de destination.
    - Pour chaque paquet correspondant, elle extrait le vecteur d'initialisation et le texte chiffré, puis déchiffre le texte à l'aide de la fonction `dechiffrer_aes_cbc`.
    - Les messages déchiffrés sont stockés dans une liste qui est ensuite renvoyée.
- Fonction `afficher_messages` :
    - Cette fonction prend une liste de messages en entrée.
    - Elle affiche chaque message de la liste avec son numéro d'ordre.

Partie 4: Un peu de recul

- Est-ce une bonne pratique de toujours utiliser la même clé ?

Dans les faits, utiliser la même clé en permanence est bien sûr loin d'être recommandé. En effet, bien que le mode actuel de chiffrement qu'utilisent Alice et Bob est très efficace, utiliser toujours la même clé n'est pas la méthode la plus sécurisée. Afin d'améliorer encore plus la protection des informations que ceux-ci s'échangent, ils devraient changer de clé le plus régulièrement possible. Effectivement, s'ils changent de clé très régulièrement, toute personne souhaitant déchiffrer leurs échangent devra d'abord trouver la clé utilisée, et ce, à chaque changement de clé. Cela peut devenir très efficace si la méthode de cassage utilisée est assez longue. En effet, le temps que le casseur trouve la clé utilisée avec des échantillons de messages, il est possible que celle-ci ait déjà été remplacée, et donc, que toute la procédure soit à recommencer.

- De quel vrai protocole réseau le protocole PlutotBonneConfidentialité est-il inspiré ?

Le protocole *PlutotBonneConfidentialité* est bien évidemment inspiré du protocole SSH en réseau. En effet, c'est le protocole qui ressemble le plus à la technique utilisée ici. Comme dans *PlutotBonneConfidentialité*, le protocole SSH nécessite que le serveur web auquel un utilisateur essaie de se connecter dispose d'un certificat obtenu suite à une demande à une autorité de certification. La technique est simple, la machine utilisateur demande au serveur un certificat, si celui-ci est valide avec la clé donnée par l'autorité de certificat, alors cela veut dire que celui-ci a bien été validé et est donc fiable. Ici, tout repose sur l'utilisation d'une clé de chiffrement et de déchiffrement comme ce que l'on peut retrouver dans le protocole PlutotBonneConfidentialité afin de vérifier si tout est sécurisé.

- Deux autres exemples où un tel protocole peut être utile

L'utilisation de tels protocoles n'est, en effet, pas utile que dans ce but. Effectivement, devoir transmettre des messages à son destinataire sans que ceux-ci ne puissent être interceptés est plutôt courant et impliquent, dans de très nombreux cas, de plus grands enjeux. Premièrement, nous pouvons bien évidemment citer l'exemple de l'utilisation de messages chiffrés lors de guerres. Il est évident que passer des informations de manière discrète et sans pouvoir être déchiffré est primordial. La majorité des techniques utilisées durant les conflits sont les chiffrements avec des clés publiques et privées comme le RSA par exemple. En effet, il est très compliqué avec les moyens techniques actuels de venir à bout de la sécurité de ces protocoles, surtout avec des clés très grandes. Deuxièmement, nous pouvons penser à l'utilisation de tels protocoles dans le domaine de la communication de tous les jours. En effet, avec moins d'enjeux que le domaine militaire mais toujours très important, des méthodes de chiffrement des informations sont utilisées dans tous les moyens de communications que nous utilisons au quotidien. Cela a pour but d'empêcher l'accès aux échanges privés.

- Connaissons-nous des applications de messagerie utilisant des mécanismes de chiffrement similaires?

Bien entendu, la plupart des messageries actuelles utilisent des méthodes de chiffrement des messages plus ou moins sécurisées. On peut prendre l'exemple de WhatsApp qui utilise le fonctionnement de chiffrement bout en bout qui est particulièrement efficace à condition que les clés publiques et privées soient stockées correctement. Le chiffrement bout en bout ressemble particulièrement au RSA puisque l'expéditeur utilise la clé publique du destinataire (préalablement partagée) pour chiffrer le message et que le destinataire le déchiffre avec sa clé privée. A plus haut niveau de chiffrement, nous pouvons citer Olvid, notamment connue pour être la nouvelle application que doit utiliser le ministère français pour ses communications textuelles par messages depuis cette fin d'année. Cette application utilise également le chiffrement bout en bout, mais de niveau "supérieur" comme elle le qualifie. C'est la seule application certifiée par l'ANSSI (Agence Nationale de Sécurité des Systèmes d'Informations).

- Nos arguments en faveur ou contre ces législations (CSAR, EARN IT Act), notamment en matière de vie privée

Le fait d'obliger les fournisseurs de services numériques à pouvoir déchiffrer les communications de leurs utilisateurs est compréhensible. En effet, ces projets de lois encouragent cette possibilité afin de pouvoir vérifier l'activité de certains utilisateurs pouvant être catégorisés de suspects voire dangereux sur Internet. En effet, si chacun peut converser avec n'importe qui sans que personne d'externe n'ait accès à ces échanges, cela peut encourager à l'utilisation de certaines applications dans l'unique but de converser à propos de sujets illégaux, à organiser des actes répréhensibles, etc… Cependant, cela nuit directement au respect de la vie privée des utilisateurs innocents. En effet, converser en ayant à l'esprit que n'importe qui n'étant pas concerné par l'échange puisse avoir accès aux informations que l'on envoie peut être très dérangeant. C'est pour cela qu'il est très difficile à notre

époque d'équilibrer la sécurité et le respect de la vie privée en ligne. D'un côté, permettre la vérification des échanges pourrait empêcher certains événements dévastateurs comme des actes terroristes pour les plus graves, mais aussi d'autres pratiques illégales comme le trafic de drogues par exemple. De l'autre côté, avoir accès à n'importe quelle conversation pourrait nuire à la discrétion et à l'anonymat (sur certains forums par exemple) de beaucoup d'utilisateurs ne souhaitant pas être espionnés ou retrouvés.

Répartition des tâches

Au niveau de la répartition des tâches, nous nous sommes répartis les différentes parties de la SAÉ et notamment les différentes questions. Au début de la SAÉ, pour être sûr qu'on ait les mêmes bases sur celle-ci, la première partie des questions a été faite en collaboration à deux. Ainsi, une fois le contexte de cette SAÉ prêt et compris par chacun, nous avons discuté de comment se répartir au mieux les tâches afin d'avancer rapidement et surtout efficacement. C'est la raison pour laquelle, nous avons choisi de se répartir suivant ce tableau :

	Partie 1	Partie 2	Partie 3	Partie 4
Quentin	double sdes / cassage astucieux	recherche de clé dans les images	recherche des messages dans la trace réseau	Aide sur les réponses aux questions
Loann	cassage brutal / expérience sur les cassages	Réponses aux questions (tests associés) / recherche de clé dans les images	Rapport sur la méthode de recherche	Réponses aux questions

Comme vous le voyez dans ce tableau ci-dessus, certaines tâches ont été choisies en collaboration car nous savions que ces dernières pourraient être potentiellement compliquées et longues. De plus, les tâches communes permettaient également que nous travaillions en collaboration constante pour discuter de nos avancées ou même de nos problèmes pour que l'autre membre nous aide facilement.