Document de conception

1.1
Quel algorithme et quelle taille de clef choisissez-vous ?
Nous choisissons d'utiliser l'algorithme AES (Advanced Encryption Standard) car il est adapté à notre situation de chiffrement de fichier, largement utilisé dans l'industrie et ce depuis presque vingt ans. Nous avons considéré les options d'utiliser le 3DES ou le Twofish, mais nous pensons qu'il est plus bénéfique pour nous de travailler avec l'AES afin de nous familiariser avec cet algorithme en premier. L'AES est également plus rapide que le 3DES et le Twofish pour la même taille de clef. Nous avons choisi une taille de clef de 128 bits car cela est suffisant pour la sécurité et possède de meilleures performances que des clefs de 192 bits ou 256 bits. Le seul intérêt d'utiliser des tailles de clefs plus conséquentes serait pour se protéger d'attaques venant d'ordinateurs quantique, pour l'instant nous n'y sommes pas encore.

Sources et justifications :
Nous avons pu voir suite à une implémentation en python des différents algorithmes (AES-3DES-Twofish) pour des tailles de clefs identiques, l'AES été le plus rapide.
Pourquoi choisir AES plutôt que Twofish(Page3)
https://www.ijser.org/researchpaper/Evolution-of-AES-Blowfish-and-Two-fish-Encryption-Algorithm.pdf

Pourquoi une telle taille de clef :
ANSSI Recommandations Partie 2.1.1

RÈGLES ET RECOMMANDATIONS :

RègleCléSym-1. La taille minimale des clés symétriques utilisées jusqu'en 2020 est de 100 bits.
RègleCléSym-2. La taille minimale des clés symétriques devant être utilisées au-delà de 2020 est de 128 bits.

RecomCléSym-1. La taille minimale recommandée des clés symétriques est de 128 bits.

Figure 1: Extrait des RGS de l'ANSSI

https://crypto.stackexchange.com/questions/20/what-are-the-practical-differences-between-256-bit-192-bit-and-128-bit-aes-enc

1.2

Quel mode d'opération choisissez-vous ?

Pour notre projet nous avons choisi d'implémenter le « CipherText Stealing ». Nous avions commencé par choisir le modèle CBC avant de passer au CTS pour des raisons que nous verrons à la question 6. Le CBC consiste à utiliser un vecteur d'initialisation (noté IV), une taille de bloc (noté N) et un algorithme de chiffrement symétrique avec une clé. Ensuite nous découpons le fichier à crypter en x éléments de taille N choisi auparavant. Le premier bloc va être mis dans un XOR avec le vecteur d'initialisation. Le résultat de cette opération va ensuite être envoyé dans notre algorithme de chiffrement et il en résultera notre premier bloc chiffré. Nous recommençons cela pour chaque bloc du fichier à un détail près : nous utilisons l'IV que pour le premier bloc, il est ensuite remplacé par le bloc chiffré précédent. Si le dernier bloc n'est pas de la taille N alors nous utilisons le padding. Pour cela nous rajoutons un 1 à la fin du bloc puis que des 0 jusqu'à obtenir un bloc de taille N. Ainsi lors du déchiffrement il ne restera plus qu'à supprimer tous les 0 et le premier 1 rencontré en partant de la fin.

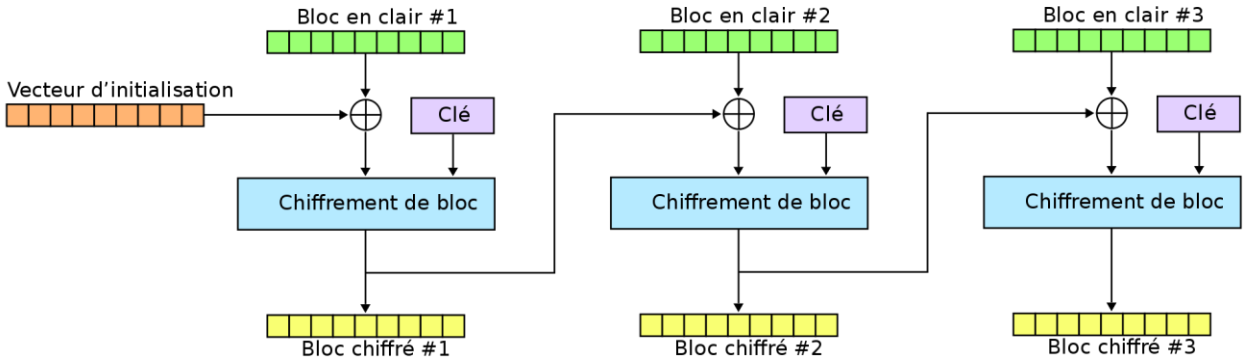


Figure 2: Schéma de chiffrement du modèle CBC

Pour le déchiffrement le principe est sensiblement le même, il est juste inversé. Il faut commencer par passer le bloc chiffré dans l'algorithme de déchiffrement puis mettre le resultat dans le un XOR avec le bloc chiffré précédent. Le premier bloc n'ayant pas de bloc chiffré avant on le met dans le XOR avec le vecteur d'initialisation qui fait partie du message chiffré envoyé.

1.3

Quel *padding* choisissez-vous ?

Pour le padding nous avons choisi le procédé de rembourrage n°2 de la norme ISO / IEC 9797-1. Ce procédé consiste simplement à rajouter un 1 suivi d'autant de 0 que nécessaire pour obtenir la taille de bloc désirée.

Nous avons choisi cette méthode car nous voulions pouvoir prendre en compte toutes tailles de fichier et non simplement un fichier d'une taille multiple de notre taille de bloc. Ainsi cela nous a semblé être la façon la plus pratique à implémenter. Le plus simple aurait été de rajouter seulement des 0, mais au moment de supprimer le *padding*, lors du déchiffrement, il aurait été compliqué de s'arrêter au bon moment. C'est-à-dire qu'en ne mettant que des 0 nous n'aurions pas été capable de déterminer si le 0 faisait partie du message ou non. Mais en ajoutant d'abord un 1 puis que des 0 il nous suffit de supprimer tous les 0 jusqu'à rencontrer un 1 de le supprimer à son tour et nous avons le texte en claire.

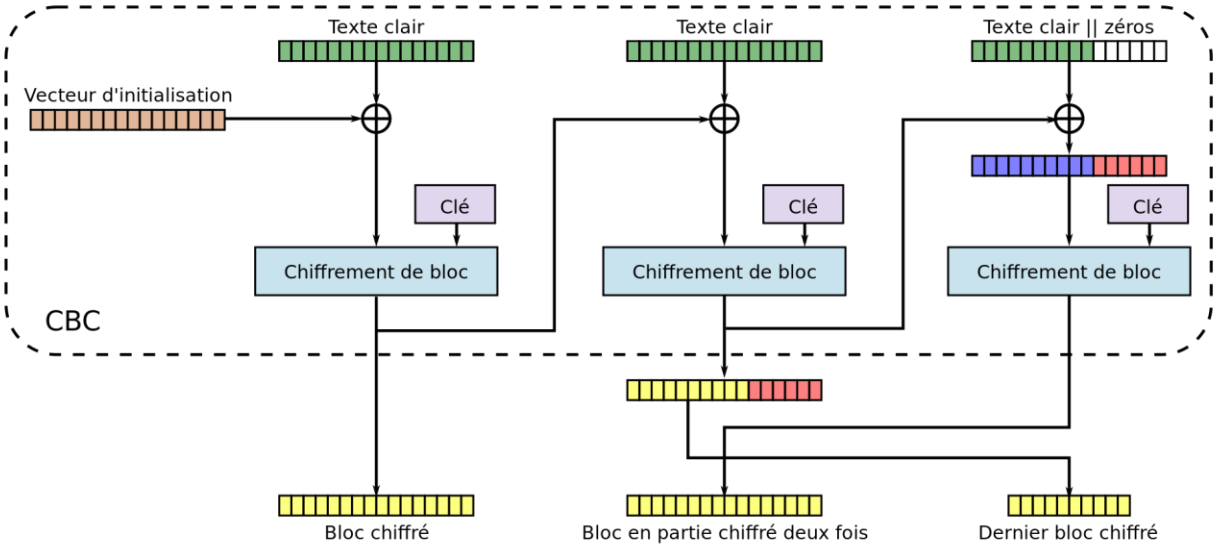
2.
Le fait de modifier le fichier chiffré modifie le déchiffrement à partir du bloc qui a été modifié mais ce n'a pas d'influence jusqu'à la fin. Cette modification a des répercussions sur le bloc chiffré modifié et celui qui suit étant donné qu'avec le modèle CBC on utilise le bloc chiffré n pour déchiffrer le bloc n+1. Le fait qu'un bloc chiffré soit modifié va fausser le résultat du déchiffrement qui va lui même être utilisé pour déchiffrer le bloc suivant. Ainsi le bloc chiffré modifié sera corrompu tout comme le bloc suivant car il utilise le bloc chiffré qui a été modifié pour être déchiffré. Cependant le bloc chiffré n+1 n'étant pas modifié, à partir du bloc n+2 le déchiffrement sera correct. Le fait d'utiliser un bloc pour en chiffrer un autre a pour effet d'augmenter l'entropie de ce système, en revanche lorsqu'il y a une modification dans les blocs chiffrés cela affecte les blocs qui suivent.

3.
Pour vérifier l'intégrité de notre message nous avons mis en place un système de XOR entre tous nos blocs chiffrés. Le résultat de ce XOR, qui a une chance sur 2^{128} d'être identique à un autre bloc de 128 bits, est stocké dans un fichier au format JSON transmis avec le message chiffré. Ce fichier JSON contient également l'IV. Ainsi si le message a été modifié il est facile de le savoir, il suffit de faire le XOR entre tous les blocs du message reçu et le comparer avec celui transmis dans le JSON. La probabilité que les modifications du cyphertext donnent le même bloc de 128 bits que l'originale étant quasiment nulles, on peut donc supposer que les deux blocs doivent être identiques sinon cela signifie que le message a été corrompu.

4.
Même si deux fichiers ont un contenue identique, avec notre code le chiffrement de ces fichiers sera différent car nous générons un vecteur d'initialisation aléatoirement et ainsi il est à chaque fois différent ce qui va influer dès le début sur le reste du chiffrement et donner un résultat différent. Il y a 2^{128} possibilités de bloc différents, il est donc presque impossible de tomber deux fois sur le même bloc et donc que les deux contenues soient chiffrés de la même façon.

5.
Notre ensemble de fichier chiffré a une taille de 517 bytes, alors que ces fichier mis sous forme d'archive au format ZIP a une taille de 337 bytes. Soit une différence de 180 bytes qui correspond à une diminution de 35% du volume initial. On voit donc que ce format donne une taille de fichier bien moindre. En effet ceci se traduit par le fait que mettre au format ZIP revient à compresser les fichiers. Le but de ce format est donc de stocker nos fichiers sous forme compressé afin de gagner de la pace sur nos disques.

6.
Pour pouvoir utiliser des fichiers de toutes tailles avec notre code nous avons décidé de passer du modèle CBC au modèle CTS. Ce modèle de chiffrement reprend celui du CBC qui est un chiffrement par bloc. Le CTS reprend le fonctionnement du CBC, cependant il ajoute un padding (décrit en question 1.3) qui va nous permettre d'adapter notre code à n'importe quel fichier quel que soit sa taille. De plus ce modèle échange et combine les deux derniers blocs. Ainsi il est nécessaire d'avoir les deux pour en déchiffrer un.



En ce qui concerne justement le déchiffrement il est un peu plus complexe que celui du CBC. Il est le même dans les deux cas sauf pour les deux dernier bloc où ils faut jongler entre les deux blocs qui ont été echangé. Pour obtenir le texte en clair il faut commencer à decoder de la même façon que pour CBC. Une fois arrivé à l'avant dernier bloc il faut le le passer dans l'algorithme de déchiffrement (on note A le resultat obtenu). Ensuite une fois dechiffré on recupère les k derniers bits qui correspondent au nombre de bits ajouté lors du padding. On rajoute ces bits au dernier bloc chiffré (on note ce bloc B). On met B dans l'algorithme de dechiffrement. On applique au resultat du déchiffrement un XOR avec l'antépénultième bloc chiffré et ainsi on obtient l'avant dernier bloc de texte clair. Pour avoir le dernier bloc de texte clair il faut faire un XOR entre A et B et on obtient le texte claire plus le padding. Pour enlever le padding il suffit, en partant de la fin, de supprimer tous les 0 jusqu'au premier 1 rencontré que l'on supprime également.