# Rapport de projet

Projet Cryptographie

« Utilitaire de chiffrement de fichiers »

*Projet réalisé en Master 1 option Cybersécurité à l’ISEN TOULON par :*

Guillaume BLANC DE LANAUTE

Valentin MAGNAN

*Projet encadré par :*

Pierre Girard



Sommaire

[Rapport de projet 1](#_Toc33825188)

[I – Abréviations 4](#_Toc33825189)

[II – Introduction 5](#_Toc33825190)

[III - Choix de l’algorithme symétrique et de la taille de clé : AES-192 6](#_Toc33825191)

[IV - Choix du mode d’opération : CBC 7](#_Toc33825192)

[V - Choix du padding : PKCS #7 8](#_Toc33825193)

[V.I - Réflexion à propos de l’implémentation du padding par la bibliothèque javax.crypto 8](#_Toc33825194)

[VI - Choix du format de fichier 9](#_Toc33825195)

[VI.I - Première approche 9](#_Toc33825196)

[VI.II - Seconde approche 9](#_Toc33825197)

[VII – Mécanisme d’intégrité 10](#_Toc33825198)

[VII.I – Implémentation du CMAC/OMAC1 10](#_Toc33825199)

[VIII – Chiffrement d’une liste de fichiers 12](#_Toc33825200)

[IX – Choix du nouveau mode de chiffrement 13](#_Toc33825201)

[IX.I - Mode : CTS 13](#_Toc33825202)

[Chiffrement 14](#_Toc33825203)

[Déchiffrement 14](#_Toc33825204)

[IX.II - Format de fichier et d’archive 14](#_Toc33825205)

[IX.III – Problème 15](#_Toc33825206)

[X – Observations et tests : 16](#_Toc33825207)

[Etape 2 : Modification du ciphertext 16](#_Toc33825208)

[Explications 16](#_Toc33825209)

[Etape 3 : Modification du ciphertext avec mécanisme d’intégrité 18](#_Toc33825210)

[Conclusion 18](#_Toc33825211)

[Etape 4 : Liste de fichiers chiffrant 2 fichiers identiques différemment 19](#_Toc33825212)

[Etape 5 : Taille de l’archive de sortie par rapport aux fichiers chiffrés 19](#_Toc33825213)

[XI – Fonctionnement des différentes versions 20](#_Toc33825214)

[XI.I – Version 1 et 2 20](#_Toc33825215)

[XI.II – Version 3 20](#_Toc33825216)

[En mode chiffrement 20](#_Toc33825217)

[En mode déchiffrement 20](#_Toc33825218)

[XI.III – Version 4 20](#_Toc33825219)

[En mode chiffrement 20](#_Toc33825220)

[En mode déchiffrement 20](#_Toc33825221)

[XII - Sources 21](#_Toc33825222)

[ANNEXE B1 DU RGS DE L’ANSSI 21](#_Toc33825223)

[ENONCE DU PROJET 21](#_Toc33825224)

[RECOMMANDATIONS NSA : 21](#_Toc33825225)

[PADDING 21](#_Toc33825226)

[CRYPTANALYSE AES 21](#_Toc33825227)

[CRYPTANALYSE DES 21](#_Toc33825228)

[INTEGRITE 21](#_Toc33825229)

[ZIP 21](#_Toc33825230)

[CTS 21](#_Toc33825231)

# I – Abréviations

ANSSI : Agence nationale de sécurité des systèmes informatiques

RGS : Référentiel général de sécurité

AES : Advanced encryption standard

DES : Data encryption standard

NSA : National security agency

CBC : Cipher block chaining

ECB : Electronic codebook

CTS : Ciphertext stealing

PKCS : Public key cryptographic standards

IV : Initialization vector

MAC : Message authentication code

HMAC : Keyed-hash message authentication code

OMAC : One-key MAC

CMAC : Cipher-based MAC

KDF : Key derivation function

HKDF : KDF based on a HMAC

OWASP : Open web application security project

API : Application programming interface

XOR : eXclusive OR function

# II – Introduction

L’objectif de ce projet est de réaliser un utilitaire de chiffrement de fichiers répondant aux spécificités indiquées par notre professeur M. GIRARD.

Les consignes sont les suivantes :

* L’utilitaire doit être conforme au Référentiel Général de Sécurité version 2 de l’ANSSI et en particulier à son annexe B1 [17TU[Source]](#_Annexe_B1_du)U17T.
* Les mécanismes implantés doivent se baser sur un algorithme de chiffrement par bloc sans fonction de hachage.

Nous avons découpé le projet en 4 grandes étapes (énoncé du projet [17T[Source]](#_Enonce_du_projet)17T) :

*Etape 1 :*

L’utilitaire s’utilisera comme une commande en ligne de la forme suivante pour chiffrer ou déchiffrer un fichier :

filecrypt –enc|-dec –key K…K –in <input file> -out <output file>

La clef est donnée en hexadécimal. Par exemple une clef de 8 bits à 1 sera donnée par –key FF. Le programme doit vérifier que le fichier d’entrée existe et que le fichier de sortie n’existe pas (ou demander la permission explicite de l’écraser).

*Etape 2 :*

Sans changer l’algorithme ni le mode de chiffrement, on décide de rajouter un mécanisme assurant l’intégrité du fichier. Pour cette question, il est nécessaire d’effectuer une recherche (hors du RGS) pour trouver comment faire cohabiter le chiffrement et l’intégrité.

*Etape 3 :*

L’utilitaire doit maintenant prendre en entrée une liste de fichiers. La sortie sera une archive compressée sous format zip des fichiers chiffrés. Il est demandé de chiffrer différemment deux fichiers au contenu identique.

*Etape 4 :*

Les fichiers chiffrés doivent être traités par des scripts existants et ceux-ci sont sensibles à la longueur des fichiers.

Il faut donc impérativement produire des fichiers chiffrés de la même longueur que les fichiers clairs (à l’octet près).

Il est possible de rajouter un fichier supplémentaire de gestion du chiffrement dans l’archive produite.

Il faut trouver un nouveau mode de chiffrement répondant à cette nouvelle contrainte (toujours compatible avec le RGS) et modifier notre format de fichier chiffré / d’archive en conséquence.

Une version du programme sera réalisée pour chacune de ces grandes étapes.

# III - Choix de l’algorithme symétrique et de la taille de clé : AES-192

Il existe différents algorithmes de chiffrement par bloc : AES, DES, triple DES, Blowfish, Serpent etc.

Tout d’abord, nous avons tout de suite évité d’utiliser l’algorithme DES, celui-ci ayant été cryptanalysé. A ce jour il est possible de casser une clé DES en 24h seulement. [17T[SOURCE]](#_CRYPTANALYSE_DES)17T

Quant au triple DES : celui-ci est vulnérable à une attaque à clair connue du fait de sa faible taille de bloc (64bits) et de sa longueur de clé effective (112bits).

Nous avons décidé de choisir l’algorithme AES car celui-ci répond aux règles et recommandations de l’ANSSI [17T[p12 annexe B1 du RGS]](#_Annexe_B1_du)17T :

*RègleAlgoBloc-2. Pour un algorithme de chiffrement dont l’utilisation au-delà de 2020 est*

*envisagée, aucune attaque nécessitant moins de Nop = opérations de calcul ne doit être*

*connue.*

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

Après plusieurs recherches, nous avons remarqué qu’une cryptanalyse d’AES a été réalisé en 2011 par des chercheurs de Microsoft : leur attaque leur permet de retrouver une clé AES 128bits en opérations, soit 4 fois plus rapide qu’en bruteforce ( opérations).

Pour suivre la règle RègleAlgoBloc-2 ainsi que la recommandation RecomCléSym-1 nous utiliserons donc des clés supérieures à 128bits (192 ou 256 pour AES) même si l’attaque découverte par Microsoft reste impraticable sur des clés de 128bits avec les technologies actuelles. [17T[1 et 2]](#_CRYPTANALYSE_AES_:)17T

*RecomAlgoBloc-1 : Il est recommandé d’employer des algorithmes de chiffrement par bloc largement éprouvés dans le milieu académique.*

Or dans la partie « Mécanisme conforme au référentiel » [p12 annexe B1 du RGS], il est écrit : « AES, tel qu’il est spécifié dans le FIPS 197, est un mécanisme de chiffrement par bloc conforme au référentiel. »

De plus, il est recommandé par la NSA, l’agence nationale de sécurité américaine assure que toute les tailles de clés d’AES sont suffisantes pour protéger des documents classifiés jusqu’au niveau SECRET, le niveau TOP SECRET quant à lui nécessite des clés de 192 ou 256bits. [17T[1 et 2]](#_RECOMMANDATION_NSA_:)17T

L’algorithme AES-192 est donc conforme aux recommandations de l’ANSSI mais également de la NSA assurant un niveau de protection très important avec des clés de 192bits.

# IV - Choix du mode d’opération : CBC

Il existe plusieurs modes d’opération de chiffrement par bloc en cryptographie : ECB, CBC, CFB, OFB, CTR, GCM…

Après consultation des 3 recommandations [17T[p13 annexe B1 du RGS]](#_Annexe_B1_du)17T, nous avons décidé de choisir le mode CBC car il les satisfait :

*RecomModeChiff-1 : L’emploi d’un mode opératoire de chiffrement non déterministe est recommandé.*

CBC est bien un mode opératoire de chiffrement non déterministe car il fait entrer en jeu un vecteur d’initialisation généré aléatoirement et permet d’obtenir 2 blocs chiffrés différents lorsque les 2 bloc d’entrée sont identiques.

*RecomModeChiff-2 : L’utilisation d’un mode opératoire de chiffrement se fera de préférence*

*conjointement à l’utilisation d’un mécanisme d’intégrité. Un tel mécanisme pourra être*

*indépendant du mode de chiffrement.*

[17T[p14 annexe B1 du RGS]](#_Annexe_B1_du)17T : « Aucun des modes classiques n’apporte la moindre protection en intégrité » (GCM quant à lui peut apporter cette protection d’intégrité).

Nous avons choisi de ne pas implémenter le mode GCM car il est bien plus complexe à implémenter que le mode CBC associé à un mécanisme d’intégrité.

*RecomModeChiff-3 : On utilisera de préférence des modes opératoires disposant d’une preuve*

*de sécurité.*

Voir « Mécanisme conforme au référentiel » [17T[p14 annexe B1 du RGS]](#_Annexe_B1_du)17T : « Le mode de chiffrement CBC utilisant une primitive de chiffrement conforme au référentiel comme l’AES […] est un mécanisme de chiffrement symétrique conforme au référentiel ».

# V - Choix du padding : PKCS #7

L’annexe B1 du RGS ne nous donne aucune règle ou recommandation pour le choix du padding.

En cryptographie par bloc, certains modes tel que CBC nécessitent une taille de bloc spécifique lors du chiffrement.

Lorsque l’on décompose le plaintext en N messages de la taille d’un bloc, le dernier message ne fait généralement pas la taille du bloc, sauf si :

On va donc devoir effectuer un padding (ou remplissage en français) afin de remplir le dernier message à chiffrer lorsqu’il ne fait pas la taille d’un bloc.

Il existe plusieurs méthodes de padding [17T[1]](#_PADDING)17T comme par exemple :

*Zero Padding*: remplit la fin du message uniquement par des bytes à 00.

*ANSSI X9.23 padding*: remplit la fin du message par des octets à 00 sauf pour le dernier qui sera la taille totale du padding.

*PKCS#5 et PKCS#7 padding* :La valeur de chaque octet ajouté est celle du nombre d’octets manquants pour compléter le bloc, par exemple 05 05 05 05 05 s’il manque 5 octets au message.

PKCS#5 padding fonctionne uniquement pour des tailles de bloc de 64bits alors que PKCS#7 fonctionne pour des blocs de taille allant jusqu’à 256bits. [17T[2]](#_PADDING)17T

La primitive AES fixant les blocs à une taille de 128bits, nous ne pouvons pas utiliser le PKCS#5 padding.

Ces méthodes de padding PKCS#5 et PKCS#7 nous semblent très efficaces, d’un point de vue sécurité : on peut voir assez facilement s’il y a eu une erreur dans le déchiffrement du message en inspectant le padding. Mais également d’un point de vue temps de calcul, le remplissage du padding nécessite uniquement de calculer le nombre de bytes manquants et d’attribuer les valeurs de ceux-ci.

Nous avons donc décidé de choisir le PKCS#7 padding qui nous semble plus fiable et efficace.

## V.I - Réflexion à propos de l’implémentation du padding par la bibliothèque javax.crypto

La classe « javax.crypto.Cipher » ne nous propose pas de PKCS#7 padding mais uniquement du PKCS#5 padding pour AES en mode CBC ce qui n’est pas logique car les blocs AES ne font pas 64bits mais 128 comme dit précédemment. [17T[3]](#_PADDING)17T

Apparemment il s’agirait d’un abus de langage, les commentaires de la réponse de Henry sur Stackoverflow nous le fait savoir. [17T[4]](#_PADDING)17T

Nous avons tout de même décidé de l’implémenter nous-mêmes.

# VI - Choix du format de fichier

Soit || l’opérateur de concaténation.

## VI.I - Première approche

Nous avons décidé de stocker l’IV à la fin de nos fichiers chiffrés afin de pouvoir le réutiliser pour le déchiffrement.

Notre fichier chiffré se présente de la façon suivante : Ciphertext || IV.

## VI.II - Seconde approche

Pour rajouter le mécanisme d’intégrité à notre utilitaire de chiffrement de fichier nous avons décidé de rajouter le MAC juste après l’IV dans notre fichier chiffré de cette façon : Ciphertext || IV || MAC

# VII – Mécanisme d’intégrité

Tout d’abord, il faut savoir qu’il existe plusieurs ordres pour faire un MAC [17T[1]](#_INTEGRITY)17T :

* *Encrypt-then-MAC* : Calcul du MAC à partir des données chiffrées puis on le concatène aux données chiffrées ;
* *Mac-then-Encrypt* : Calcul du MAC à partir des données non chiffrées puis on le concatène aux données non chiffrées et on chiffre le tout ;
* *Encrypt-and-MAC* : Calcul du MAC à partir des données non chiffrées puis on chiffre les données et ensuite concatène le MAC aux données chiffrées.

Nous avons décidé d’utiliser la méthode Encrypt-then-MAC car c’est la seule méthode permettant à la fois l’intégrité du ciphertext et du plaintext [17T[2]](#_INTEGRITY)17T, mais aussi car c’est la seule méthode où le MAC ne donne aucune information sur le plaintext (car calculé à partir du ciphertext).

Les algorithmes MAC peuvent être basés sur 2 primitives cryptographiques différentes [17T[3]](#_INTEGRITY)17T:

1. Fonction de hachage cryptographique : par exemple pour faire un HMAC
2. Chiffrement par bloc : pour calculer un OMAC, CBC-MAC, PMAC…

Premièrement nous avions implémenté un HMAC, ce qui était interdit dans les consignes du projet.

Nous avons donc dû changer pour un MAC basé sur du chiffrement par bloc afin de respecter la consigne suivante : « Les mécanismes implantés doivent se baser sur un algorithme de chiffrement par bloc sans fonctions de hachage ».

Notre premier choix a été le CBC-MAC mais celui-ci n’est pas sûr, en effet :

« Une attaque décrite par [Preneel](https://fr.wikipedia.org/wiki/Bart_Preneel" \o "Bart Preneel) et Van Oorschot nécessite environ  textes connus accompagnés de leurs codes d'authentification (n est la longueur du bloc en bits), elle permet de forger des codes d'authentification. CBC-MAC n'est pas sûr avec des messages de taille variable. » [17T[4]](#_INTEGRITY)17T

Nous avons donc changé d’avis pour un OMAC1 qui est une amélioration du XCBC-MAC (étant lui-même une amélioration du CBC-MAC [17T[5, p2 partie 4]](#_INTEGRITY)17T).

OMAC1 est équivalent à CMAC et est recommandé par le NIST depuis 2005. [17T[6]](#_INTEGRITY)17T

## VII.I – Implémentation du CMAC/OMAC1

Il faut savoir que nous avons besoin d’une clé secrète pour calculer le CMAC, or réutiliser la clé de chiffrement pour l’intégrité est d’un point de vue sécurité une mauvaise idée.

C’est pourquoi nous avons utilisé un HKDF afin de dériver notre clé passée en paramètre lors de l’exécution de notre utilitaire de chiffrement :

On a défini notre clé passée en paramètre comme la clé « Master » d’une taille de 24 octets.

Nous dérivons la clé « Master » en 2 sous clés : integrityKey et encKey.

encKey est la clé permettant le chiffrement des données, celle-ci fait 24 octets (pour du AES-196)

integrityKey est la clé permettant l’intégrité des données, nous avons choisi une taille de clé de 16 octets, la taille de clé pour un CMAC peut-être variable.

Et enfin nous calculons le CMAC des données suivantes : données chiffrées || IV.

On le concatène à la fin du fichier chiffré pour pouvoir le récupérer facilement pour la vérification d’intégrité lors du déchiffrement.

Nous avons utilisé 2 bibliothèques pour implémenter notre mécanisme d’intégrité :

Pour utiliser le HKDF, nous avons utilisé la bibliothèque « hkdf de patrickfav » disponible sur github : 17T<https://github.com/patrickfav/hkdf>

Cette bibliothèque passe tous les tests de l’OWASP Dependency-Check qui est un utilitaire permettant de vérifier que la bibliothèque n’est sujette à aucune vulnérabilité recensée dans les bases de données du NIST.

La seconde bibliothèque utilisée est en fait une collection d’API utilisées en cryptographie, c’est « Bouncy Castle », elle nous permet de calculer le CMAC.

# VIII – Chiffrement d’une liste de fichiers

Il nous est demandé de chiffrer une liste de fichiers puis de stocker les fichiers chiffrés dans une archive au format zip.

De plus 2 fichiers identiques appartenant à cette liste doivent être chiffrés différemment.

Afin de répondre à cette problématique, nous avons décidé de chiffrer chaque fichier avec une clé différente.

Appelons clé master la clé passée en paramètre au programme.

Pour ce faire nous générons N « sous-clé master » (N étant le nombre de fichier) dérivées à partir de la clé master et des IV correspondant à chaque fichier.

Ces N clés nous permettent ensuite de générer pour chaque fichier les sous clés encKey et integrityKey.

Les fichiers auront donc leurs propres clés uniques et seront de ce fait chiffrés différemment même si plusieurs fichiers sont identiques.

Voir étape 4 dans la partie VIII du rapport pour la vérification du fonctionnement.

# IX – Choix du nouveau mode de chiffrement

Dans la partie 4 du projet, il nous est demandé de trouver un nouveau mode de chiffrement satisfaisant les contraintes suivantes :

* Les fichiers chiffrés doivent faire la même taille que les fichiers clairs
* Compatible avec le RGS

Il nous est autorisé de créer un fichier supplémentaire de gestion de chiffrement.

Le format de fichier et d’archive sont également à modifier.

## IX.I - Mode : CTS

Nous avons décidé d’opter pour le mode CTS ou CipherText Stealing.

Ce n’est pas tout à fait un mode de chiffrement à part entière, en effet CTS est applicable à des modes tel que ECB, CBC…

Il fonctionne de la manière suivante :

On chiffre les données selon notre mode de chiffrement préféré et compatible avec CTS (dans notre cas CBC), jusqu’à arriver à l’avant-dernier bloc à chiffrer.

On chiffre l’avant-dernier bloc puis on identifie le nombre d’octets manquants dans le dernier bloc et au lieu de faire un padding on copie la fin de l’avant-dernier bloc chiffré dans le dernier bloc (après avoir effectué un XOR).

L’avant-dernier bloc chiffré moins les octets copiés correspondra au dernier bloc chiffré.

Le chiffrement du dernier bloc correspondra à l’avant-dernier bloc chiffré.

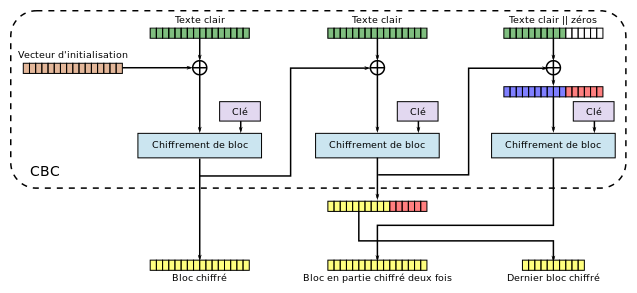
L’avant-dernier bloc aura donc une partie qui aura été chiffrée 2 fois.

Le déchiffrement se fait de la manière inverse.

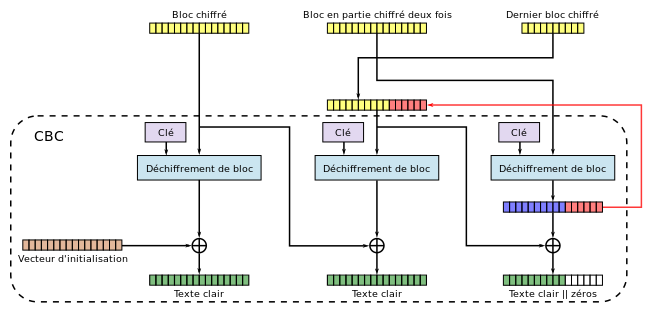
Ce mode nous a donc permis d’éviter d’avoir à faire un padding et donc nos données chiffrées et nos données claires font bien la même taille.

Ci-dessous les schémas explicatifs du chiffrement et du déchiffrement [17T[Source]](#_CTS)17T.

### Chiffrement



### Déchiffrement



CTS ne fait pas apparaître de plaintext dans les données chiffrées, seulement un bloc est chiffré presque 2 fois complètement. Il ne fait pas intervenir de problème de sécurité supplémentaire au mode de chiffrement sur lequel il est basé.

CBC étant compatible avec le RGS, l’implémentation de CTS avec CBC reste compatible avec le RGS.

## IX.II - Format de fichier et d’archive

Sachant que nous devons produire des fichiers chiffrés de la même taille que les fichiers clairs, notre nouveau format de fichier chiffré en est donc réduit au plus simple, c’est-à-dire que l’on a uniquement les données chiffrées dans le fichier.

Ayant toujours besoin de notre mécanisme d’intégrité et d’un IV pour chaque fichier, nous avons créé un fichier gestionnaire de chiffrement appelé « EncryptData ».

Il contient les IV et MAC de chaque fichier sur une seule ligne de cette manière :

IV1 || MAC1 || IV2 || MAC2 || … || IVN || MACN

## IX.III – Problème

Nous avons rencontré un problème qui n’est pas résolu à ce jour.

Le mode CTS a besoin d’un minimum de 1 bloc entier pour pouvoir effectuer le CTS car le principe de ce dernier s’appuie sur l’utilisation 2 blocs ou 1 bloc et l’IV.

Il est donc impossible de chiffrer un fichier d’une taille inférieur à un bloc soit inférieur à 16 octets dans notre cas.

# X – Observations et tests :

## Etape 2 : Modification du ciphertext

Pour étudier les modifications du plaintext lors du déchiffrement d’un ciphertext altéré nous avons réalisé 5 tests différents :

#### Test n°1 : Modification d’un bit au début du ciphertext

Ligne 00000000h de l’éditeur hexadécimal, on remplace le premier octet 1F par 0F

Résultat lors du déchiffrement : on observe une modification dans les 16 premiers caractères du fichier, les 16 premiers caractères sont illisibles.

#### Test n°2 : Modification d’un bit au milieu du ciphertext

Ligne 00000180h, on remplace le premier octet BD par BC

Résultat lors du déchiffrement : on observe également une altération sur un bloc de 16 caractères au milieu du fichier, ils sont aussi illisibles.

#### Test n°3 : Modification d’un bit à la fin du ciphertext

Ligne 00000360h, on remplace le dernier octet 8D par 0D

Résultat lors du déchiffrement : On observe une modification du plaintext au début du fichier, dans les 16 premiers caractères, cela semble étrange mais il y a une explication.

#### Test n°4 : Modification d’un bit presque à la fin du ciphertext

Ligne 00000350h, on remplace le dernier octet 1F par 3F

Résultat lors du déchiffrement : on observe cette fois ci une modification presque complète des 16 derniers caractères du plaintext à la fin du fichier.

#### Test n°5 : Modification de 3 octets dans le ciphertext

Nous avons modifié 3 octets aléatoirement dans le ciphertext.

Résultat lors du déchiffrement : on observe 3 zones totalement altérées dans le plaintext.

### Explications

Lors des tests 1 à 4, nous avions modifié uniquement 1 bit du ciphertext.

Utilisant le mode CBC, en modifiant une partie d’un bloc du ciphertext on va altérer le déchiffrement de ce bloc mais également du bloc suivant car pour obtenir le plaintext du bloc suivant, on effectue un XOR avec le bloc précédent dans le ciphertext (qui a été altéré).

Quand la modification d’un bloc du ciphertext est uniquement d’un bit alors le bloc de plaintext associé est altéré et le bloc suivant peut également l’être dans une moindre mesure car il s’agit uniquement d’un bit modifié dans un XOR (= une modification possible dans le plaintext du bloc suivant).

Les résultats des tests 1,2 et 4 sont bien cohérents avec le fonctionnement du déchiffrement en mode CBC : 

Le résultat du test 3 quant à lui semble étrange, pourquoi une modification d’un bit à la fin du fichier chiffré modifie un tout petit morceau du début du fichier ?

L’explication est tout à fait logique, en effet nous avons choisi le format de fichier suivant : {ciphertext,IV} donc lorsqu’on modifie une partie des 16 derniers octets du fichier chiffré, on modifie en fait l’IV et non le ciphertext.

Or on applique un XOR entre le résultat du déchiffrement du premier bloc de ciphertext et l’IV, qui a été modifié, on a donc bien la modification d’un caractère du premier bloc du plaintext.

Enfin, lors du test n°5, on a bien 3 parties du plaintext qui ont été altérées.

Nos observations sont donc tout à fait conformes à ce que l’on aurait pensé en utilisant CBC.

#### Informations complémentaires :

Nous nous sommes servis de l’éditeur hexadécimal « UltraEdit » pour faire nos modifications.

Les fichiers ayant servi aux tests sont dans le dossier « Fichiers étape 2 » du rendu de projet.

## Etape 3 : Modification du ciphertext avec mécanisme d’intégrité

Nous avons effectué plusieurs tests pour vérifier notre implémentation du mécanisme d’intégrité :

#### Test 1 :

On chiffre l’image ASUS.jpg, l’image chiffrée est appelée : imgCrypte.jpg

On a créé une copie de cette image que l’on a ensuite modifié, cette image chiffrée modifiée a pour nom : imgCrypteMod.jpg

On déchiffre l’image imgCrypte.jpg dans decrypt.jpg : tout se passe bien, l’image est bien la même que l’originale, aucune erreur n’est relevée.

Enfin on tente de déchiffrer l’image chiffrée modifiée : une exception de type FileIntregrityException est levée et nous explique : « Fichier corrompu : Le MAC calculé ne correspond pas au MAC récupéré ».

#### Test 2 :

Cette fois-ci nous essayons avec des fichiers textes.

On chiffre in.txt en out.txt

On modifie out.txt dans outm.txt

De même on tente de déchiffrer out.txt : le fichier est bien déchiffré.

Puis on essaye de déchiffrer outm.txt : FileIntegrityException est à nouveau levée.

### Conclusion

100% des fichiers chiffrés modifiés sont détectés comme corrompus.

Tous les fichiers chiffrés non modifiés sont déchiffrés.

Notre mécanisme d’intégrité est bel est bien fonctionnel.

#### Informations complémentaires :

Les fichiers ayant servi aux tests sont dans le dossier « Fichier étape 3 » du rendu de projet.

## Etape 4 : Liste de fichiers chiffrant 2 fichiers identiques différemment

Nous chiffrons 3 fichiers :

File1.txt et file2.txt qui sont identiques ainsi que img.jpg.

Nous décompressons ces 2 fichiers de l’archive vers le dossier « result file1 file2 encrypted », on remarque qu’ils sont bien chiffrés différemment.

Nous les déchiffrons ensuite de la même façon que nous les avions chiffrés.

Ils contiennent bel et bien toujours les mêmes données.

Notre solution est donc conforme à la contrainte suivante : « Il est demandé de chiffrer différemment deux fichiers au contenu identique. ».

#### Informations complémentaires :

Les fichiers ayant servi aux tests sont dans le dossier « Fichier étape 4 et 5 » du rendu de projet.

## Etape 5 : Taille de l’archive de sortie par rapport aux fichiers chiffrés

En réutilisant les fichiers utilisés dans l’étape 4, nous remarquons que la taille des fichiers chiffrés n’a pas été compressée, de plus l’archive est plus lourde que la somme des tailles des fichiers chiffrés.

En effet les algorithmes de compression se basent sur des similitudes/motifs, or les fichiers chiffrés doivent sembler les plus aléatoires possibles, il n’est donc pas possible de les compresser.

Pour expliquer l’augmentation des tailles des fichiers chiffrés dans l’archive, il faut se référer au format zip : chaque fichier appartenant à un fichier zip stocke un fichier d’en-tête supplémentaire contenant des informations tel que des commentaires, la taille du fichier d’origine, le nom du fichier…

“Each entry stored in a ZIP archive is introduced by a local file header with information about the file such as the comment, file size and file name, followed by optional "extra" data fields, and then the possibly compressed, possibly encrypted file data.” [17T[Source]](#_ZIP)17T

Il est donc tout à fait normal que la somme des tailles des fichiers chiffrés soit inférieure à la taille de l’archive zip.

#### Informations complémentaires :

Les fichiers ayant servi aux tests sont dans le dossier « Fichier étape 4 et 5 » du rendu de projet.

# XI – Fonctionnement des différentes versions

Le projet comprend un total de 4 versions différentes suivant les [17T4 grandes étapes du projet](#_I_-_Introduction)17T.

Pour chaque version, la commande à exécuter est de la forme :

java -jar release-X.jar -enc|-dec -key K..K -in <input file> -out <output file>

Pour toutes les versions :

* la clé doit être écrite au format hexadécimal sur 24 octets soit 48 caractères.
* -enc pour chiffrer, -dec pour dechiffrer.

Ce qu’il faut mettre dans -in et -out peut différer selon les versions, voici les explications ci-dessous.

## XI.I – Version 1 et 2

Il est possible uniquement de chiffrer/déchiffrer un seul fichier.

Il suffit d’ajouter le fichier d’entrée après -in et le fichier de sortie après -out sans omettre les extensions des fichiers si elles existent.

## XI.II – Version 3

Il est possible cette fois de chiffrer/déchiffrer plusieurs fichiers.

### En mode chiffrement

Il faut écrire la liste des fichiers après -in.

La sortie sera une archive compressée, il suffit de mentionner son nom après -out sans ajouter l’extension .zip, celle-ci est automatiquement rajoutée par l’utilitaire.

### En mode déchiffrement

Il faut décompresser les fichiers du zip obtenu après chiffrement.

Passer la liste des fichiers que vous souhaitez déchiffrer après -in.

La sortie se fait de la même façon qu’en mode chiffrement.

## XI.III – Version 4

Dans cette version il n’est possible de chiffrer uniquement des fichiers d’une taille supérieure ou égale à 16 octets.

### En mode chiffrement

De même que pour la version 3

### En mode déchiffrement

Cette fois, au lieu de passer en paramètre d’entrée la liste des fichiers décompressés, il faut passer en paramètre le fichier zip.

La sortie sera un dossier contenant les fichiers déchiffrés utilisant le nom donné après -out.

# XII - Sources

## ANNEXE B1 DU RGS DE L’ANSSI

17T<https://www.ssi.gouv.fr/uploads/2014/11/RGS_v-2-0_B1.pdf>

## ENONCE DU PROJET

[17Thttp://bit.ly/2VrzcNy](http://bit.ly/2VrzcNy)17T

## RECOMMANDATIONS NSA :

1. <https://web.archive.org/web/20070927035010/http://www.cnss.gov/Assets/pdf/cnssp_15_fs.pdf>
2. <https://fr.wikipedia.org/wiki/Advanced_Encryption_Standard#Recommandations_de_la_NSA>

## PADDING

1. <https://en.wikipedia.org/wiki/Padding_(cryptography)#Block_cipher_mode_of_operation>
2. <https://crypto.stackexchange.com/questions/9043/what-is-the-difference-between-pkcs5-padding-and-pkcs7-padding/9044#9044>
3. <https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html>
4. <https://stackoverflow.com/questions/20770072/aes-cbc-pkcs5padding-vs-aes-cbc-pkcs7padding-with-256-key-size-performance-java/20770158#20770158>

## CRYPTANALYSE AES

1. <https://www.theregister.co.uk/2011/08/19/aes_crypto_attack/>
2. <https://fr.wikipedia.org/wiki/Advanced_Encryption_Standard#Attaques_sur_la_version_compl%C3%A8te>

## CRYPTANALYSE DES

<https://fr.wikipedia.org/wiki/Data_Encryption_Standard#Attaques>

## INTEGRITE

1. <https://en.wikipedia.org/wiki/Authenticated_encryption>
2. <https://crypto.stackexchange.com/questions/202/should-we-mac-then-encrypt-or-encrypt-then-mac/205#205>
3. <https://en.wikipedia.org/wiki/Message_authentication_code#Implementation>
4. <https://fr.wikipedia.org/wiki/CBC-MAC#S%C3%A9curit%C3%A9>
5. <http://abcdrfc.free.fr/rfc-vf/pdf/rfc3566.pdf>
6. <https://en.wikipedia.org/wiki/One-key_MAC>

## ZIP

<https://en.wikipedia.org/wiki/Zip_(file_format)>

## CTS

<https://fr.wikipedia.org/wiki/Mode_d%27op%C3%A9ration_(cryptographie)#Chiffrement_avec_vol_de_texte_:_%C2%AB_CipherText_Stealing_%C2%BB_(CTS)>