Rapport de stage

Vincent CAUJOLLE

Table des matières

Table des figures

Liste des tableaux

Glosaire

1. XOR ($\oplus$) :

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

TABLE 1 – table de vérité du XOR

2. byte : 8 bits

1 AES

Toutes les informations qui m'ont permis d'écrir cette section sont directement tiré de [1].

Face au manque de standard de chiffrement par block, NIST (National Institute of Standards and Technology) a lancé un concours en 1997 pour créer un nouveau standard de chiffrement : AES (Advance Encryption Strandard).

De là naît un nouvel algorithme inspiré de la proposition des cryptographes belge Joan Daemen et Vincent Rijmen. Il permet de chiffrer des blocks de 128 bits et existe sous trois variante :

nom	taille de la clef (bits)	taille des blocks (bits)	nombre de rounds
AES 128	128	128	10
AES 192	192	128	12
AES 256	256	128	14

TABLE 2 – Variantes d'AES

A partir de maintenant, pour fluidifier la lecture, AES 128 = AES)

Remarque : à l'origine cet algorithme pouvait supporter des blocks de 128, 192 et 256 bits. Cette fonctionnalité n'à malheureusement pas été conservé par la NIST. Cette fonction aurait permis à AES d'être post quantique.

AES permute successivement son entrée (un block de 128 bits) avec une même permutation Π (sauf au dernier round ou on utilse $\hat{\Pi}$). Cette permutation est inversible, ce qui permet de déchiffrer la sortie en remontant le processus d'AES.

On XOR la sortie de chaque round avec une clef k_i (calculé à partir de la clef secrète de 128 bits). L'avantage du XOR est que $\text{XOR}^2 = \text{id}$ (voir table 1). Donc il suffit de réutiliser XOR pour remonter AES.

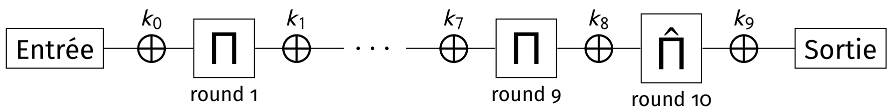


FIGURE 1 – fonctionement d'AES

1.1 Permutation Π

Cette permutation est propre à AES (ne dépend pas de la clef secrète et peut être calculé à l'avance. Cela permet d'avoir un processus de chiffrement très efficace). Elle est le résultat de la composition de 3 sous-permutations (toutes inversible). Pour mieux comprendre comment ces 3 sous-permutations marchent, il faut réorganiser le block de 128 bits en une matrice de taille 4×4 où chaque cellule contient un byte.

$$s_0|s_1|\dots|s_{15} \longrightarrow \begin{array}{|c|c|c|c|}\hline s_0 & s_1 & s_2 & s_3 \\ \hline s_4 & s_5 & s_6 & s_7 \\ \hline s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \\ \hline \end{array}$$

FIGURE 2 – Réorganisation du block

1. `SubBytes` :
   On applique à chaque cellule du block une permutation $S : \{0,1\}^8 \longrightarrow \{0,1\}^8$ (d'un byte vers un autre).

$$\begin{array}{|c|c|c|c|}\hline s_0 & s_1 & s_2 & s_3 \\ \hline s_4 & s_5 & s_6 & s_7 \\ \hline s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \\ \hline \end{array} \xrightarrow{\texttt{SubBytes}} \begin{array}{|c|c|c|c|}\hline \tilde{s}_0 & \tilde{s}_1 & \tilde{s}_2 & \tilde{s}_3 \\ \hline \tilde{s}_4 & \tilde{s}_5 & \tilde{s}_6 & \tilde{s}_7 \\ \hline \tilde{s}_8 & \tilde{s}_9 & \tilde{s}_{10} & \tilde{s}_{11} \\ \hline \tilde{s}_{12} & \tilde{s}_{13} & \tilde{s}_{14} & \tilde{s}_{15} \\ \hline \end{array}$$

FIGURE 3 – Effet de `SubBytes` sur le block, avec $\tilde{s} = S(s)$

RAJOUTER DEF S

2. `ShiftRows` :
   Cette permutation va pour chaque colone déplacer de manière cyclice ses élément de tel manière que la colone i subira le cycle

$$(0 \quad (1+i)\%4 \quad (2+i)\%4 \quad (3+i)\%4)$$

$$\begin{array}{|c|c|c|c|}\hline s_0 & s_1 & s_2 & s_3 \\ \hline s_4 & s_5 & s_6 & s_7 \\ \hline s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \\ \hline \end{array} \xrightarrow{\texttt{ShiftRows}} \begin{array}{cccc} s_0 & s_1 & s_2 & s_3 \\ s_5 & s_6 & s_7 & s_4 \\ s_{10} & s_{11} & s_8 & s_9 \\ s_{15} & s_{12} & s_{13} & s_{14} \end{array}$$

FIGURE 4 – Effet de `ShiftRows` sur le block

3. `MixColumns` :
   Pour cette permutation, on calcul dans $GF\left(2^8\right)$ (muni du polynome iréductible $x^8 + x^4 + x^3 + x + 1$ ie 100011011) le produit matriciel (à gauche) de notre block par :

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

   avec les éléments de cette matrices à comprendre comme des éléments de $GF\left(2^8\right)$

Donc, en résumé :

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{bmatrix} \xrightarrow{\Pi} \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} \tilde{s}_0 & \tilde{s}_1 & \tilde{s}_2 & \tilde{s}_3 \\ \tilde{s}_5 & \tilde{s}_6 & \tilde{s}_7 & \tilde{s}_4 \\ \tilde{s}_{10} & \tilde{s}_{11} & \tilde{s}_8 & \tilde{s}_9 \\ \tilde{s}_{15} & \tilde{s}_{12} & \tilde{s}_{13} & \tilde{s}_{14} \end{bmatrix}$$

1.2 Permutation $\hat{\Pi}$

Généralement on préfère utiliser $\hat{\Pi}$ au lieu de Π au dernier round pour avoir un algorithme de déchiffrement quasiment identique que celui de chiffrement. Avec $\hat{\Pi}$ défini tel que :

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{bmatrix} \xrightarrow{\hat{\Pi}} \begin{bmatrix} \tilde{s}_0 & \tilde{s}_1 & \tilde{s}_2 & \tilde{s}_3 \\ \tilde{s}_5 & \tilde{s}_6 & \tilde{s}_7 & \tilde{s}_4 \\ \tilde{s}_{10} & \tilde{s}_{11} & \tilde{s}_8 & \tilde{s}_9 \\ \tilde{s}_{15} & \tilde{s}_{12} & \tilde{s}_{13} & \tilde{s}_{14} \end{bmatrix}$$

C'est exactement Π mais sans la permutation `MixColumns`.

1.3 Création des clefs k_i

A partir d'une clef secrète k (de 128 bits) il faut créer une série de clefs $k_0 \dots k_{10}$. Pour ça on sépare cette clef en 4 mots de 32 bits (4 byte) chacun.

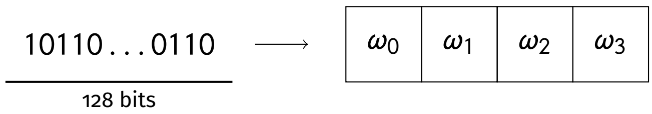


FIGURE 5 – réorganisation des clefs en suite de mots de 32 bits

On définit la première clef $k_0 = \boxed{\omega_{0,0}|\omega_{0,1}|\omega_{0,2}|\omega_{0,3}} = k$ (ie k_0 est égale à la clef secrète). Ensuite, on calcul $k_i = \boxed{\omega_{i,0}|\omega_{i,1}|\omega_{i,2}|\omega_{i,3}}$ en fonction de k_{i-1} tel que :

$$\forall i \in \{1,2,3\} \quad \begin{cases} \omega_{i,0} & = \omega_{i-1,0} \oplus g_i(\omega_{i-1,3}) \\ \omega_{i,j} & = \omega_{i-1,j} \oplus \omega_{i,j-1} \quad \forall j \in \{1,2,3\} \end{cases}$$

Avec $g : \{0,1\}^{32} \to \{0,1\}^{32}$ une fonction tirée des standards d'AES.

2 AES CBC

Toutes les informations qui m'ont permis d'écrir cette section sont directement tiré de [2].

A lui seul, AES ne permet de chiffrer que des blocks de 128 bits. On pourrait se dire qu'il suffit d'appliquer AES sur l'ensemble des blocks (c'est le mode ECB).

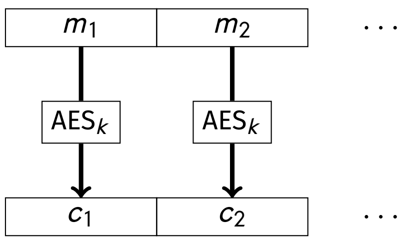
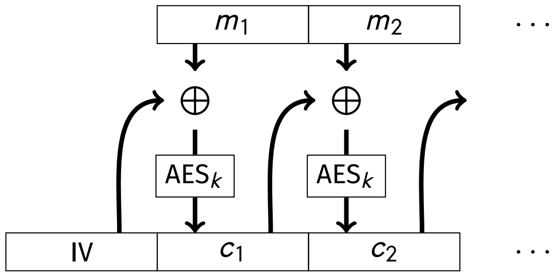


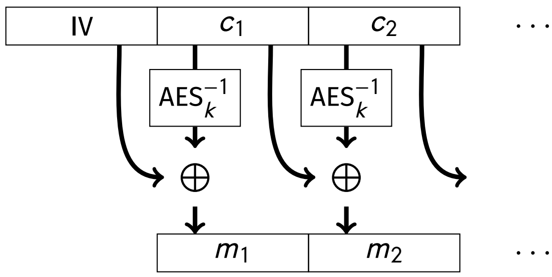
FIGURE 6 – Fonctionement d'AES ECB, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

Mais, si on utilise la même clef sur plusieurs blocks, le chiffré deviens est très facile à déchiffrer. Il faudrait donc autant de clefs que de blocks pour assurer un chiffrement sécurisé. Cela doublerai la taille du message chiffré et n'est donc pas utilisé.

Il faut donc rajouter une couche supplémentaire pour pouvoir utilser AES afin de chiffrer ce que l'on veut. On s'intéresse ici au mode CBC (Cipher Block Chaining). Son fonctionnement est décrit sur la figure 7



(a) Chiffrement



(b) Déchiffrement

FIGURE 7 – Fonctionement d'AES CBC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

On remarque que le premier block du chiffré est un Initialization Vector (IV). C'est un block choisit aléatoirement. Ensuite, un deuxième point important de ce mode d'utilisation d'AES (ou de tout autre méthode de chiffrement par block) est qu'il faut que la taille du message en clair soit exactement un multiple de 128 bits. La solution pour palier à ce problème est de rajouter un padding. C'est à dire que l'on complète le message pour que sa taille soit exactement égale au prochain multiple de 128 bits (si le message fait est déjà un multiple de 128 bits on rajoute 16 byte de padding). Il peut prendre plein de forme différente, on peut compléter avec des 0, des bits aléatoires... La seul règle importante est que le dernier byte du block doit contenir le nombre de byte de padding qui ont été rajouté (entre 0 et 15).

Pour ma part mon padding suivra la règle suivante : tous les bytes du padding ont la même valeurs (donc celle du denier byte). Par exemple s'il manque 3 byte à mon message pour être un multiple de 16 byte (i.e 128 bites) alors je rajoute à la fin du message $\boxed{02|02|02}$. Si c'est déjà un multiple de 16 byte je rajoute à la fin

$\boxed{15|15|15|15|15|15|15|15|15|15|15|15|15|15|15|15}$.

J'avais essayé d'implémenter la padding classique (remplir de 0, par exemple pour un padding de 3 bytes cela donne $\boxed{00|00|02}$) mais je devais mal implémenter quelque chose car je n'arrivais pas à me munir d'un padding oracle (voir 2.2) alors que la théorie veux que ça ne change rien.

2.1 Mon implémentation

Dans le cade de ma mission je dois implémenter AES CBC en RUST. J'ai choisit de ne pas réimplémenter AES car j'avais du mal à implémenter des éléments relatif à g (voir 1.3) mais aussi car une implémentation d'AES est très régulièrement inutilisable en réalité car trop facile à attaquer. Rien que des informations tels que la consomation électrique de l'ordinateur, le temps qu'il met à répondre ou encore le champ électromagnétique qu'il émet sont suffisant pour casser mon implémentation.

J'ai donc implémenté le mode CBC avec le padding énnoncé à la fin de 2. J'ai choisit pour simplifier la tache de me restreindre à chiffrer des chaines de charactères. De là est apparu un pseudo problème, le type Char en RUST peut faire de 1 à 4 byte. Mais,au déchiffrage, ne connaissant pas le nombre de byte associé à chaque charactères, j'ai été dans l'obligation de faire comme si chaque charactère ne faisait qu'un byte. Donc si mon message n'est pas uniquement écris avec des charactères ASCI (1 byte) alors j'aurais de la perte d'information.
Par exemple si je chiffre et déchiffre le message : "Je vais à l'école" j'obiendrai un message du type "Je vais $£ l'&ùcole".

J'ai choisit d'ignorer ce problème étant donné qu'il n'est du qu'au format de l'entrée.

2.2 Pading oracle attack

Dans cette section, j'explique comment j'ai attaqué AES CBC et de manière équivalente tout les modes de chiffrement non autentifié (qui ne fournissent pas une preuve d'autenticité du chiffré) et qui utilisent un padding.

L'attaque que j'ai mené repose sur un principe simple. Pour chaque algorithme de chiffrement qui utillisent un padding, n'importe qui pourra toujours avoir accès à un padding oracle (On n'a encore jamais trouvé de contre exemple).

Avoir accès à un padding oracle c'est pouvoir déduire du comportement du serveur si le padding du message que je lui demande de déchiffrer est correct (on peut par exemple déduire cette information du temps que mets le serveur à nous répondre). Je vais montré ci dessous qu'à partir du moment ou l'on a accès à ce genre d'information on peut casser le chiffré.

2.2.1 Déchiffrer un block

En reprenant la figure 7 et en l'adaptant pour un block c on obtien la figure 8.

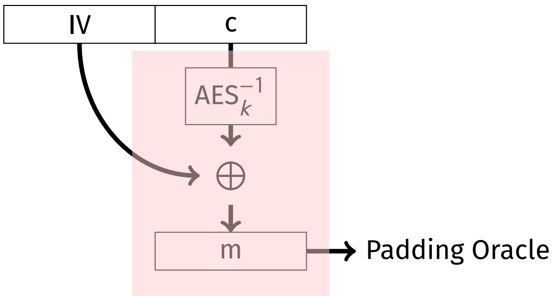


FIGURE 8 – Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible)

Le but de l'attaque est de trouver un IV tel que le padding du block déchiffré m est valide. En effet, si on arrive à trouver un IV tel que $m = \boxed{15|\ldots|15}$ alors

$$\mathrm{IV} \oplus \boxed{15|\ldots|15} = \mathrm{AES}_k^{-1}(\mathrm{c})$$

On aura donc réussi à déchiffrer c ! Pour arriver à trouver cet IV on procède de la manière suivante :

On pose IV = $\boxed{00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|b}$ avec b un byte quelconque. On fait varirer b (256 possibilités) jusqu'à obtenir une réponse positive du padding oracle. Une fois arrivé à cette étape il existe 2 situation dans lesquelle on peut se trouver :

1. Dans le cas le plus courant on aura m = $\boxed{??|??|\ldots|??|00}$. C'est ce qu'on cherche !
2. Mais il se peut aussi que l'avant dernier byte permette à plus d'un padding d'être correcte. En effet, si ce byte vaut 01 alors il existe un byte b tel que l'IV produise un m = $\boxed{??|??|\ldots|??|01|01}$.

Ce deuxième cas est génant mais il est facile de vérifier dans quel cas on se trouve en modifiant l'avant dernier byte de l'IV. En effet si en modifiant l'avant dernier byte de l'IV produit une réponse favorable du padding oracle cela signifie qu'on est dans le premier cas sinon on est dans le deuxième.

$$\boxed{??|??|\ldots|42|00} \longrightarrow \text{padding valide}$$
$$\boxed{??|??|\ldots|42|01} \longrightarrow \text{padding invalide}$$

Une fois à cette étape on peut créer un IV qu'on appelle ZIV (zeroing IV). Cet IV permet de mettre à 0 les bytes que l'on change dans m. Dans notre cas on a directement

IV = ZIV.

On sait maintenant comment créer le premier ZIV, supposons que l'on soit capable de créer le n-ième ZIV : ZIV_n (cet IV assure que les n derniers bytes de m valent 0). Essayons de trouver ZIV_{n+1}.

On pose IV $=$ ZIV_n + $\boxed{00|\ldots|00|b|n|\ldots|n}$ avec n bytes $\boxed{n}$ et b, un byte quelconque. Cet IV permet d'avoir m $=$ $\boxed{??|??|\ldots|??|n|\ldots|n}$ car 0 est l'élément neutre de $\oplus$ dans $\mathbb{Z}/2\mathbb{Z}$.
Comme à la première étape (celle qui sert à trouver le premier ZIV), on fait varier b (256 possiblités) jusqu'à avoir une réponse positive du padding oracle. En effectuant la même vérification qu'à la première étape, on est en mesure de trouver b tel que m $=$ $\boxed{??|??|\ldots|??|n|\ldots|n|}$ avec cette fois ci n+1 byte $\boxed{n}$. Alors on remarque que ZIV_{n+1} $=$ IV $\oplus$ $\boxed{00|\ldots|00|n|\ldots|n}$ car $m \oplus \boxed{00|\ldots|00|n|\ldots|n} = \boxed{??|\ldots|??|00|\ldots|00}$.

Ainsi, on est en mesure de trouver un IV (ZIV_{16}) tel que $m = \boxed{00|\ldots|00}$ et donc comme on l'a dit en début de section et comme on peut le voir sur la figure 9 on peut déchiffrer le block c car si $a \oplus b = 0$ alors $a = b$ donc $\text{AES}_k^{-1}(\text{c}) = \text{ZIV}_{16}$.

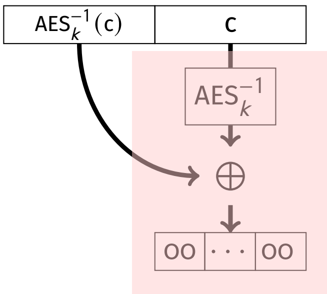


FIGURE 9 – Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible)

2.2.2 Généralisation à n blocks

On rappelle le shéma de déchifrement d'AES CBC sur la figure 10. On remarque alors que si on est en mesure de trouver $\text{AES}_k^{-1}(c_i)$ alors on est cappable de trouver m_i $\forall i$. En effet $\text{AES}_k^{-1}(c_i) \oplus c_{i-1} = m_i$. Donc en déchiffran chaque block avec une attack par padding oracle on peut déchiffrer l'entierté du chiffré.

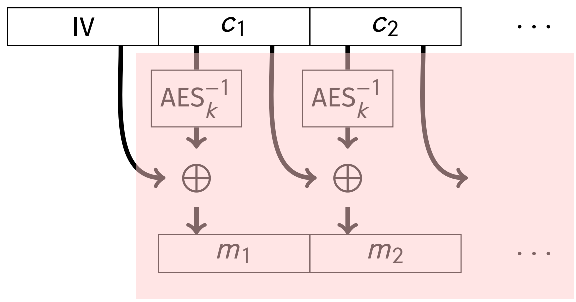


FIGURE 10 – Déchiffrement d'AES CBC avec en rouge ce qui ne nous est pas accessible

3 AES GCM

Toutes les informations qui m'ont permis d'écrir cette section sont directement tiré de [2].

On s'est intéressé dans la section précédente au mode CBC en montrant qu'AES CBC était facilement attaquable et donc pas fiable. On s'intéresse ici au mode GCM (Galois Counter Mode), le mode d'utilisation le plus courant d'AES. Comme pour AES CBC, AES GCM fait des opérations sur des blocks de 128 bits pour permettre de chiffrer de grands fichier. Mais, à la différence d'AES CBC, AES GCM utilise un algorithme d'authentification en parallle du chiffrement ce qui permet d'assurer que le chiffré n'a pas été changé. Cette deuxième partie est essentielle car il a été montré qu'un algorithme de chiffrement non authentifié est quasiment systématiquement attaquable.

3.1 CTR

Comme CBC, CTR (Counter) est un mode de manipulation des blocks qui permet de chiffrer des données avec des algorithme de chiffrement par blocks (comme AES) tout en garantisant un certain niveau de sécurité. CTR est le mode utilisé dans GCM mais sans l'authentification.Il est assez similaire à CBC mais à comme gros avantage de ne pas obliger de rajouter un padding à la fin des données clairs.

Son fonctionement est décrit sur la figure 11. *Le* $+1$ *signifie qu'on incrémente de 1 l'IV généré aléatoirement.*

3.2 GMAC

Pour obtenir GCM il faut rajouter une étape d'authentification. Elle est fait par un algorithme incrémental GMAC (Galois Message Authentification Code). Cet algorithme produit au cours du chiffrement des blocks un Tag qui résume de manière sécurisé toutes les informations sur la donné à chiffré. L'intéret de cela est que chaque entrée

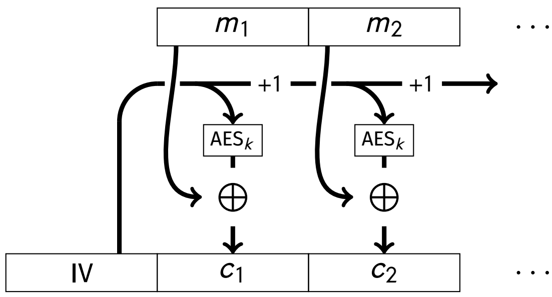


FIGURE 11 – Fonctionnement d'AES CTR, avec AES$_k$ le chiffrement d'un bloc par AES avec la clef secrète k

produira un Tag unique. Donc si quelqu'un a modifié le chiffré, alors il y aura une erreur lors du déchiffrement.Cela permet de garantir que la donné que vous déchiffrez est bien celle que l'on souhaite récupérer.

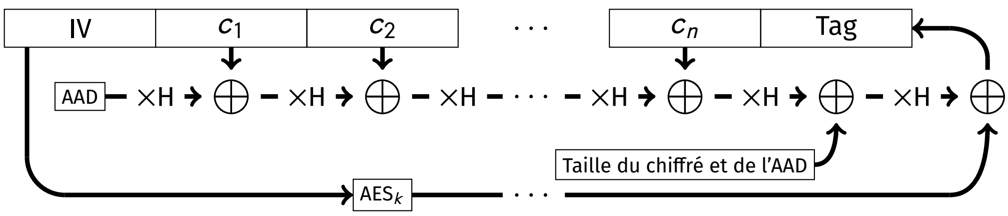


FIGURE 12 – Fonctionnement du GMAC, avec AES$_k$ le chiffrement d'un bloc par AES avec la clef secrète k, et ×H la multiplication dans $GF\left(2^{128}\right)$ par H = AES$_k$ $\left(0^{128}\right)$.

Dans la figure 12, une nouvelle entrée, appelée AAD (Additional Authenticated Data), est introduite. L'AAD est une donnée que l'on souhaite authentifier, mais pas nécessairement chiffrer. Par exemple, imaginons que vous envoyez un mot de passe accompagné d'une adresse web. Vous voudriez chiffrer le mot de passe pour le protéger, tout en laissant l'adresse en clair. Cependant, si l'adresse n'est pas authentifiée, un attaquant pourrait la modifier, et vous ne le sauriez pas lors du déchiffrement. L'AAD permet de protéger contre ce risque en garantissant que l'adresse n'a pas été altérée."

3.3 GCM

GCM est la combinaison du mode CTR (voir 3.1) et de la création d'un Tag avec GMAC (voir 3.2). Combiné à AES, cela nous donne l'algorithme de chiffrement authentifié le plus largement utilisé. Son fonctionement est décrit sur la figure 13.
Pour résumer, AES GCM prend 4 entrées :

1. un nonce (ou IV) généré aléatoirement (que l'on choisit de mettre au début du chiffré pour réduir le nombre d'objets à stoquer).

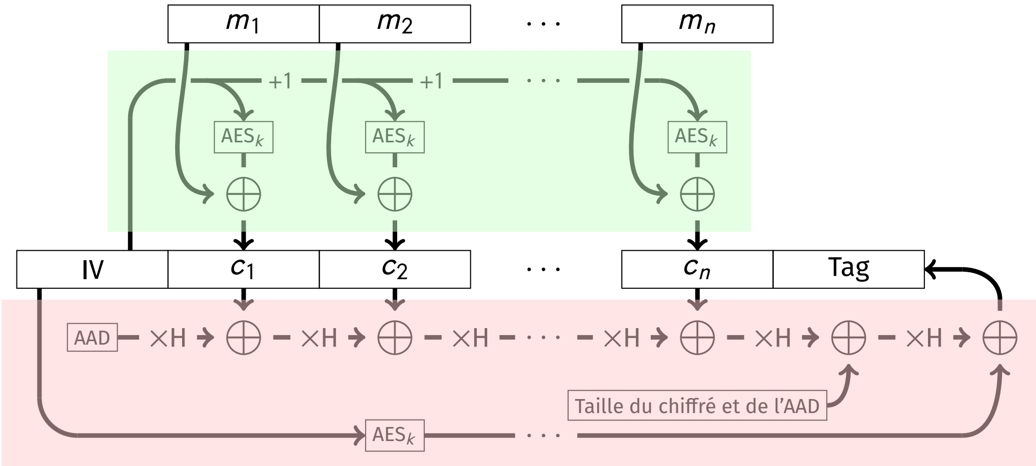


FIGURE 13 – Fonctionnement d'AES GCM avec en vert CTR et en rouge GMAC

2. une clef (de préférence généré aléatoirement)
3. des données à chiffrer et authentifier
4. des données à authentifier (on peut aussi ne pas en fournir)

et fournit 2 sorties :

1. Le chiffré de la donné à chiffrer
2. Le tag associé aux entrées (mis à la fin du chiffré)

4 SOFTHSM

5 Signature

5.1 RSA

5.2 Courbe elliptiques avec Ed25519

6 Mise en place d'un programme de chiffrement/déchiffrement

6.1 Contexte

Maintenant que j'ai pris connaissance des algorithmes pour chiffrer et signer des données, je dois mettre en place une solution écrite en Rust qui permettra de chiffrer et de déchiffrer des fichiers. Cette solution devra permettre d'utiliser soit AES-GCM, soit ChaCha20-Poly1305 (un autre algorithme de chiffrement non détaillé précédemment).

Une optimisation que je pourrais envisager serait de proposer une version qui chiffrerait "au fil de l'eau", c'est-à-dire de manière progressive, bloc par bloc, à mesure que les données sont lues ou écrites. Cette méthode présente l'avantage de maintenir un coût constant en espace mémoire, car elle ne nécessite pas de charger l'intégralité du fichier en mémoire avant de le chiffrer.

Pour exécuter ce programme de chiffrement, il suffira d'utiliser les commandes suivantes :

```
cat enc.data | ./rudecrypt options > dec.data
```

Les options permettront de choisir l'algorithme utilisé. Pour le moment peuvent soit valoir "aes" pour AES-GCM, ou bien "cha" pour ChaCha20-Poly1305 ou encore "aes_stream" pour la version au fil de l'eau d'AES-GCM.

le programme de chifrement s'appelle rucrypt et celui de déchifrement rudecrypt

6.2 Mise en place

Cette fois-ci, contrairement à Ed25519, je n'ai pas tout reprogrammé. J'ai décidé d'utiliser des crates de confiance, largement éprouvées. Pour les identifier, je me suis aidé d'un site qui référence toutes les solutions couramment utilisées pour la cryptographie en Rust [3]. J'utilise AES 256 (voir le tableau 2). Pour chaque algorithme, je dois donc générer, avant le chiffrement, une clé de 32 octets (256 bits) et un nonce (vecteur d'initialisation) de 12 octets (96 bits). Il est démontré que cacher le nonce n'a aucun intérêt en termes de sécurité. Je place donc le nonce en clair avant le texte chiffré.

On pourrait s'arrêter là, mais il est possible d'ajouter un niveau de sécurité supplémentaire. Actuellement, le chiffrement et le déchiffrement se font avec une seule clé (c'est-à-dire une clé symétrique). Le problème de cette symétrie est que toute personne ayant accès à cette clé peut lire le message en clair. Dans le cadre de ma mission, cela représente une potentielle faille. En effet, le but final de ce programme est de permettre la création de sauvegardes chiffrées de données. Il est donc préférable d'utiliser une clé asymétrique (une pour le chiffrement et une pour le déchiffrement). Dans ce cas de figure, tout le monde peut effectuer une sauvegarde de ses fichiers, mais seules les personnes autorisées pourront les récupérer, cela permet de garder le contrôle sur les flux. Pour cette raison, je génère une clé privée et une clé publique RSA, avec laquelle je chiffre ma clé symétrique. Ainsi, seule la personne en possession de la clé privée pourra déchiffrer le document. Pour plus de simplicité, je place la clé symétrique chiffrée avant le nonce.

$$\text{rucrypt(data)} \longrightarrow \boxed{\text{RSA(clef symétrique) | nonce | data chiffré}}$$

Références

[1] Dan Boneh et Victor Shoup. *A Graduate Course In Applied Cryptography*. 2023.

[2] Mike Rosulek. *The Joy of Cryptography*. 2021.

[3] *Showcase of notable cryptography libraries developed in Rust*. URL : `cryptography.rs`.