
M1 Informatique
2022

Rapport

# Attaque par saturation sur l'AES

**Réalisé par :**

Clément LYONNET
Éloïse MERCADO GARCIA
Vincent XAVIER

**Proposé par :** Mme Christina BOURA

# Table Des Matières

# 1 Introduction

## 1.1 Qu'est-ce que AES ?

L'algorithme de chiffrement AES (Advanced Encryption Standard) est un algorithme de chiffrement par bloc. Il s'agit d'un algorithme à clé symétrique, ce qui signifie que les détenteurs d'une même clé secrète sont en mesure de chiffrer et de déchiffrer des messages en se servant de cette même clé.
Il existe trois sortes d'AES :

- AES-128 : fonctionnant avec 10 tours et une clé de 128 bits.
- AES-192 : fonctionnant avec 12 tours et une clé de 192 bits.
- AES-256 : fonctionnant avec 14 tours et une clé de 256 bits.

C'est aujourd'hui l'algorithme le plus sûr à utiliser dans le contexte symétrique. Il a été créé par *Joan Daemen* et *Vincent Rijmen*, et a été standardisé en 2001. La meilleure attaque connue à ce jour contre l'AES est la recherche exhaustive, avec une complexité de $2^{128}$ en ce qui concerne l'AES-128 à 10 tours.

## 1.2 Langage utilisé

Le choix d'un langage de programmation pour l'implémentation d'un chiffrement et d'une attaque est décisif. En effet, ceci va déterminer certains paramètres comme l'efficacité et la simplicité du programme.

### 1.2.1 Problématiques

Implémenter un système de chiffrement implique de considérer plusieurs paramètres :

- Manipuler des messages d'entrées (ici nos blocs d'entrées sont fixés à 128 bits)
- Générer des clés pseudo-aléatoires
- Rapidité de chiffrement/déchiffrement
- Manipuler des bits aisément
- Optimiser la mémoire nécessaire lors de l'exécution

### 1.2.2 Choix final

En prenant en considération les points énoncés ainsi que nos connaissances en matière de langage de programmation, nous avons décidé de programmer en langage C.
En effet, celui-ci nous permet d'avoir une approche de la représentation de l'octet s'approchant de la réalité d'un système de chiffrement comme l'AES. De plus, le mode CTR étant utilisé comme mode de notre AES, nos connaissances dans ce langage nous as permis de paralléliser ce mode d'opération.

## 1.3 Mode d'opération

Comme énoncé plus haut, le mode d'opération de notre AES que nous avons choisi de mettre en place est le mode CTR.
Le mode CTR (Counter), a été inventé et proposé par *Whitfield Diffie* et *Martin Hellman* en 1979. Voici les quelques points le définissant :

- Tout d'abord, un vecteur d'initialisation (IV) de 128 bits est généré aléatoirement.
- Il y a autant de tours de CTR que de multiples de 128 bits du message clair
- Au $1^{er}$ tour, l'algorithme de l'AES prendra en entrée : IV $\oplus$ *counter* et $k$ comme clé.

- À chaque chiffrement effectué, *counter* sera modifié par une fonction qui produit une séquence sans répétition. Cependant, un simple incrément de 1 à chaque tour est suffisant et est une technique amplement utilisée : c'est cette solution qui a été retenue.
- Tous les tours de CTR peuvent se dérouler en parallèle et c'est sur ce point que ce mode d'opération est efficace
- Une fois tous les tours effectués, la sortie de CTR va être XORée avec les blocs de message clairs (précédemment complétés par des 0 afin d'avoir des blocs de 8 octets)

### 1.4 Attaque par saturation sur l'AES

Le but de ce projet est de mener une attaque par saturation sur un AES-128 simplifié. Cette version de l'algorithme comporte 4 tours au lieu de 10, afin de faire apparaitre des propriétés qui nous permettent de compromettre la sûreté de l'AES.

L'objectif de cette attaque est de retrouver la clé utilisée pour le chiffrement. Pour ce faire, l'attaquant doit disposer d'au moins 256 couples clair-chiffré utilisant une même clé : celle que nous voulons retrouver. Ces couples clair-chiffré permettent d'effectuer une cryptanalyse, que nous verrons plus en détail dans une section ultérieure.

## 2 Structure de l'AES-128 10 tours

### 2.1 Algorithme principal

Dans un premier temps, présentons l'algorithme principal de l'AES-128 à 10 tours :

**Algorithm 1** AES 10 rounds

**Require:** K (clé, 128 bits), M (clair, 128 bits)
**Ensure:** $C = AES_K(M)$ ▷ C est le chiffré
  $SubKeys[11] \leftarrow KeySchedule(K)$ ▷ Cadencement de clé → 11 sous-clés
  $S \leftarrow M$
  $S \leftarrow S \oplus AddRoundKey[0]$
  **for** $i \leftarrow 1$ *to* 10 **do** ▷ AES à 10 tours
    $S \leftarrow SubBytes(S)$
    $S \leftarrow ShiftRows(S)$
    **if** $i \leq 9$ **then**
      $S \leftarrow MixColumn(S)$
    **end if**
    $S \leftarrow S \oplus AddRoundKey[i]$
  **end for**
  $C \leftarrow S$
  **return** $C$

Décrivons en détail l'algorithme de chiffrement symétrique de l'*Advanced Encryption Standard* (AES). Il est composé de 4 fonctions principales, qu'on expliquera un peu plus tard dans cette partie :

- `SubBytes` ($S$-box),
- `ShiftRows`,
- `MixColumns`,

- `AddRoundKey`.

L'AES-128 sera représenté sous forme de tableau 4 ×4 de 8 bits chaque bloc (1 octet). Il est donc décrit en 16 octets ($8 \times 16 = 128$).

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

Un algorithme de *cadencement de clé* est calculé à partir de la clé $K$. Cette clé est également de 128 bits et aussi représentée comme un tableau de $4 \times 4$. Cet algorithme produit 4 clés pour les 4 tours (10 pour l'AES à 10 tours).

Le message à chiffrer est mis sous la forme d'un tableau $4 \times 4$ comme expliqué ci-dessus, par 16 blocs de 1 octet chacun. L'AES applique successivement sur ce tableau les opérations citées précédemment.

## 2.2 AddRoundKey

L'opération *AddRoundKey* s'effectue par bloc, un bloc du message avec un bloc de la clé qui lui correspond.

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

$\oplus$

| | | | |
|---|---|---|---|
| $k_{0,0}$ | $k_{0,1}$ | $k_{0,2}$ | $k_{0,3}$ |
| $k_{1,0}$ | $k_{1,1}$ | $k_{1,2}$ | $k_{1,3}$ |
| $k_{2,0}$ | $k_{2,1}$ | $k_{2,2}$ | $k_{2,3}$ |
| $k_{3,0}$ | $k_{3,1}$ | $k_{3,2}$ | $k_{3,3}$ |

$=$

| | | | |
|---|---|---|---|
| $a \oplus k_{0,0}$ | $a \oplus k_{0,1}$ | $a \oplus k_{0,2}$ | $a \oplus k_{0,3}$ |
| $a \oplus k_{1,0}$ | $a \oplus k_{1,1}$ | $a \oplus k_{1,2}$ | $a \oplus k_{1,3}$ |
| $a \oplus k_{2,0}$ | $a \oplus k_{2,1}$ | $a \oplus k_{2,2}$ | $a \oplus k_{2,3}$ |
| $a \oplus k_{3,0}$ | $a \oplus k_{3,1}$ | $a \oplus k_{3,2}$ | $a \oplus k_{3,3}$ |

On effectue un XOR $\oplus$ bit à bit du message avec la clé. Pour chaque tour, on utilise une clé différente, obtenue par le cadencement de clé, expliqué plus tard.

## 2.3 SubBytes

L'opération *SubBytes* est la partie substitution. Chaque octet est substitué par un octet différent.

Dans cette partie, on va s'intéresser à comment est représenté un octet et comment se passe les opérations addition (+) et multiplication (×) entre octet.

### 2.3.1 Qu'est ce qu'un octet ?

Un octet est composé de 8 bits. Un bit n'a que 2 valeurs possibles : "0" ou "1". Un octet peut donc avoir 256 valeurs possibles ($2^8$), allant de 00000000 pour représenter 0 à 11111111 pour représenter 255.

Pour traduire un octet en polynôme, on peut voir chaque chiffre comme un index. Donc un octet $A$ ayant pour valeur possible de $[\![0, 255]\!]$ tel que $A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) \in \{0, 1\}^8$ est représenté par un polynôme de degré $\leq 7$ tel que $A(x) = \sum_{i=0}^{7} a_i x^i$ avec $a_i \in \{0, 1\}$.

Exemple : Pour l'octet 137, $137 = 128 + 8 + 1$.
Il est composé de 128 ($= 2^7$), de 8 ($= 2^3$) et de 1 ($= 2^0$).
Donc l'octet 137 est représenté $10001001 \leftrightarrow (X^7 + X^3 + X^0)$.

### 2.3.2 L'addition

Pour l'addition, c'est équivalent à un XOR, c'est à dire à une addition mod 2, une addition dans $Z/_2Z$.
Exemple : On a 137 et 92. On veut faire la somme de ces octets. On procède d'abord à leur conversion. On a donc : $137 \leftrightarrow 10001001$ et $92 \leftrightarrow 01011100$.

$$\begin{array}{rrcr} & 10001001 & \leftrightarrow & 137 \\ \oplus & 01011100 & \leftrightarrow & 92 \\ \hline & 11010101 & \leftrightarrow & 213 \end{array}$$

On obtient 11010101 qui est égal à 213.

Si on utilise la notation expliquée plus haut, on a $137 \leftrightarrow (X^7 + X^3 + 1)$ et $92 \leftrightarrow (X^6 + X^4 + X^3 + X^2)$.

En faisant la somme de ces 2 polynômes, on obtient :

$$\begin{aligned} (X^7 + X^3 + 1) + (X^6 + X^4 + X^3 + X^2) &= (X^7 + X^6 + X^4 + X^3 + X^3 + X^2 + 1) \\ &= (X^7 + X^6 + X^4 + X^2 + 1) \\ &= 11010101 \leftrightarrow 213. \end{aligned}$$

On obtient bien le même résultat qu'avec le XOR.

### 2.3.3 La multiplication

On a deux octets $a$ et $b$, tous deux représentés comme des polynômes de degré $\leq 7$ (le polynôme $A(X)$ pour l'octet $a$ et le polynôme $B(X)$ pour l'octet $b$).

La multiplication est faite de cette manière :

$$A(X) \times B(X) \bmod I(X) \tag{1}$$

avec le polynôme irréductible $I(X) = X^8 + X^4 + X^3 + X + 1$;

### 2.3.4 La transformation SubBytes

*SubBytes* opère indépendamment sur chacun des 16 octets en utilisant la table de substitution ($S$-box). Elle est définie de la façon suivante :

$$\begin{array}{rccl} S: & \{0,1\}^{256} & \longrightarrow & \{0,1\}^{256} \\ & a & \longmapsto & \begin{cases} a^{-1} & \text{si } a \neq 0, \\ 0 & \text{si } a = 0, \end{cases} \end{array}$$

Pour obtenir $SubBytes(a)$, on calcule :

$$SubBytes(a) = A \times a \oplus B$$

où $A$ est une matrice $8 \times 8$ à coefficients dans [0,1] et $B \in (2)^8$:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \quad \text{et} \quad a = \begin{pmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{pmatrix}$$

À la place de faire ce calcul pour chaque octet, il suffit d'utiliser la représentation par table de la $S$-box :

| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0$a$ | 0$b$ | 0$c$ | 0$d$ | 0$e$ | 0$f$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 63 | 7$c$ | 77 | 7$b$ | $f2$ | 6$b$ | 6$f$ | $c5$ | 30 | 01 | 67 | 2$b$ | $fe$ | $d7$ | $ab$ | 76 |
| 10 | $ca$ | 82 | $c9$ | 7$d$ | $fa$ | 59 | 47 | $f0$ | $ad$ | $d4$ | $a2$ | $af$ | 9$c$ | $a4$ | 72 | $c0$ |
| 20 | $b7$ | $fd$ | 93 | 26 | 36 | 3$f$ | $f7$ | $cc$ | 34 | $a5$ | $e5$ | $f1$ | 71 | $d8$ | 31 | 15 |
| 30 | 04 | $c7$ | 23 | $c3$ | 18 | 96 | 05 | 9$a$ | 07 | 12 | 80 | $e2$ | $eb$ | 27 | $b2$ | 75 |
| 40 | 09 | 83 | 2$c$ | 1$a$ | 1$b$ | 6$e$ | 5$a$ | $a0$ | 52 | 3$b$ | $d6$ | $b3$ | 29 | $e3$ | 2$f$ | 84 |
| 50 | 53 | $d1$ | 00 | $ed$ | 20 | $fc$ | $b1$ | 5$b$ | 6$a$ | $cb$ | $be$ | 39 | 4$a$ | 4$c$ | 58 | $cf$ |
| 60 | $d0$ | $ef$ | $aa$ | $fb$ | 43 | 4$d$ | 33 | 85 | 45 | $f9$ | 02 | 7$f$ | 50 | 3$c$ | 9$f$ | $a8$ |
| 70 | 51 | $a3$ | 40 | 8$f$ | 92 | 9$d$ | 38 | $f5$ | $bc$ | $b6$ | $da$ | 21 | 10 | $ff$ | $f3$ | $d2$ |
| 80 | $cd$ | 0$c$ | 13 | $ec$ | 5$f$ | 97 | 44 | 17 | $c4$ | $a7$ | 7$e$ | 3$d$ | 64 | 5$d$ | 19 | 73 |
| 90 | 60 | 81 | 4$f$ | $dc$ | 22 | 2$a$ | 90 | 88 | 46 | $ee$ | $b8$ | 14 | $de$ | 5$e$ | 0$b$ | $db$ |
| $a0$ | $e0$ | 32 | 3$a$ | 0$a$ | 49 | 06 | 24 | 5$c$ | $c2$ | $d3$ | $ac$ | 62 | 91 | 95 | $e4$ | 79 |
| $b0$ | $e7$ | $c8$ | 37 | 6$d$ | 8$d$ | $d5$ | 4$e$ | $a9$ | 6$c$ | 56 | $f4$ | $ea$ | 65 | 7$a$ | $ae$ | 08 |
| $c0$ | $ba$ | 78 | 25 | 2$e$ | 1$c$ | $a6$ | $b4$ | $c6$ | $e8$ | $dd$ | 74 | 1$f$ | 4$b$ | $bd$ | 8$b$ | 8$a$ |
| $d0$ | 70 | 3$e$ | $b5$ | 66 | 48 | 03 | $f6$ | 0$e$ | 61 | 35 | 57 | $b9$ | 86 | $c1$ | 1$d$ | 9$e$ |
| $e0$ | $e1$ | $f8$ | 98 | 11 | 69 | $d9$ | 8$e$ | 94 | 9$b$ | 1$e$ | 87 | $e9$ | $ce$ | 55 | 28 | $df$ |
| $f0$ | 8$c$ | $a1$ | 89 | 0$d$ | $bf$ | $e6$ | 42 | 68 | 41 | 99 | 2$d$ | 0$f$ | $b0$ | 54 | $bb$ | 16 |

Pour avoir la nouvelle valeur de l'octet, il faut regarder la première moitié de l'octet en question, parcourir le tableau verticalement jusqu'à trouver la valeur, puis en regardant la deuxième moitié de l'octet, lire horizontalement et ainsi trouver le nouvel octet.
Exemple : Résultats de $S$-box de l'octet 58 : $S$-box(58) = 6a.

On procède ainsi pour toutes les cases de notre tableau $4 \times 4$.

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

$\longrightarrow$

| | | | |
|---|---|---|---|
| $S$-box($a_{0,0}$) | $S$-box($a_{0,1}$) | $S$-box($a_{0,2}$) | $S$-box($a_{0,3}$) |
| $S$-box($a_{1,0}$) | $S$-box($a_{1,1}$) | $S$-box($a_{1,2}$) | $S$-box($a_{1,3}$) |
| $S$-box($a_{2,0}$) | $S$-box($a_{2,1}$) | $S$-box($a_{2,2}$) | $S$-box($a_{2,3}$) |
| $S$-box($a_{3,0}$) | $S$-box($a_{3,1}$) | $S$-box($a_{3,2}$) | $S$-box($a_{3,3}$) |

Chaque valeur d'octet va être transformée en une autre valeur.

## 2.4 ShiftRows

*ShiftRows* applique une permutation circulaire vers la gauche aux lignes du tableau, respectivement de 0, 1, 2, 3 cases:

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
| $a_{2,0}$ | $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

$\longrightarrow$

| | | | |
|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{0,3}$ |
| $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ | $a_{1,0}$ |
| $a_{2,2}$ | $a_{2,3}$ | $a_{2,0}$ | $a_{2,1}$ |
| $a_{3,3}$ | $a_{3,0}$ | $a_{3,1}$ | $a_{3,2}$ |

## 2.5 MixColumns

La transformation *MixColumns* transforme indépendamment chacune des 4 colonnes de l'état. Elle utilise la matrice ci-dessous pour obtenir la nouvelle colonne.

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

Le résultat est le produit matriciel avec une colonne $i$ de notre tableau $4 \times 4$ et la matrice ci-dessus.

$$\begin{pmatrix} a_{0,i} \\ a_{1,i} \\ a_{2,i} \\ a_{3,i} \end{pmatrix} \times \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} b_{0,i} \\ b_{1,i} \\ b_{2,i} \\ b_{3,i} \end{pmatrix} \quad \text{avec } i \in [0,3] \text{ représentant la colonne}$$

On répéte le processus pour chaque colonne (On le fait donc 4 fois).
Exemple : Calculons

$$MixColumn \begin{pmatrix} \text{F2} \\ \text{0A} \\ 22 \\ \text{5C} \end{pmatrix}$$

Nous avons donc

$$\begin{pmatrix} \text{F2} \\ \text{0A} \\ 22 \\ \text{5C} \end{pmatrix} \times \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

avec

$$\begin{cases} b_0 = 2 \bullet \text{F2} \oplus 3 \bullet \text{0A} \oplus 1 \bullet 22 \oplus 1 \bullet \text{5C} \\ b_1 = 1 \bullet \text{F2} \oplus 2 \bullet \text{0A} \oplus 3 \bullet 22 \oplus 1 \bullet \text{5C} \\ b_2 = 1 \bullet \text{F2} \oplus 2 \bullet \text{0A} \oplus 3 \bullet 22 \oplus 1 \bullet \text{5C} \\ b_3 = 1 \bullet \text{F2} \oplus 1 \bullet \text{0A} \oplus 2 \bullet 22 \oplus 3 \bullet \text{5C} \end{cases}$$

Commençons par calculer $b_0$.

Faisons les conversions nécessaires :
L'octet F2 (11110010 en binaire) $= X^7 + X^6 + X^5 + X^4 + X$, sous forme polynôme.
0A (00001010) $= X^3 + X^1$.
22 (00100010) $= X^5 + X$.
5C (01011100) $= X^6 + X^4 + X^3 + X^2$.
2 (00000010) $= X$ , 3 (00000011) $= X + 1$, 1 (00000001) $= 1$.

$$\begin{aligned} 2 \bullet F2 &= (X) \times (X^7 + X^6 + X^5 + X^4 + X) \\ &= X^8 + X^7 + X^6 + X^5 + X^2 \\ &= X^7 + X^6 + X^5 + X^4 + X^3 + X^2 + X + 1 \\ &= 11111111 \end{aligned}$$

$$\begin{aligned} 3 \bullet 0A &= (X + 1) \times (X^3 + X^1) \\ &= X^4 + X^2 + X^3 + X^1 \\ &= 00011110 \end{aligned}$$

$$\begin{aligned} 1 \bullet 22 &= (1) \times (X^5 + X) \\ &= X^5 + X \\ &= 00100010 \end{aligned}$$

$$\begin{aligned} 1 \bullet 22 &= (1) \times (X^6 + X^4 + X^3 + X^2) \\ &= X^6 + X^4 + X^3 + X^2 \\ &= 01011100 \end{aligned}$$

On fait un XOR de ce qu'on a obtenu :

$$\begin{array}{rl} & 11111111 \\ \oplus & 00011110 \\ \oplus & 00100010 \\ \oplus & 01011100 \\ \hline & 10011111 \end{array}$$

10011111 donne 9F. Donc $b_0 = 9F$.

On procède ainsi pour les octets suivants ("0A", "22" et "5C"). On obtient donc :

$$MixColumn \begin{pmatrix} F2 \\ 0A \\ 22 \\ 5C \end{pmatrix} = \begin{pmatrix} 9F \\ DC \\ 58 \\ 9D \end{pmatrix}$$

## 2.6 KeySchedule

KeySchedule nécessite une clé maître de 128 bits. Il génère au total 11 sous-clés pour un AES de 10 tours. Les clés obtenues sont de 128 bits, que ce soit pour l'$AES-128$, l'$AES-192$ et l'$AES-256$. Les sous-clés sont également représentées comme un tableau $4 \times 4$.

KeySchedule utilise 3 principales opérations :

- `RotWord`,
- `SubWord`,
- `Rcon`

Ces opérations s'exécutent sur des colonnes. Pour faciliter la compréhension de l'algorithme, une colonne sera notée $k[i]$ avec $i \in [\![0, 3]\!]$

| | | | |
|---|---|---|---|
| $k_{0,0}$ | $k_{0,1}$ | $k_{0,2}$ | $k_{0,3}$ |
| $k_{1,0}$ | $k_{1,1}$ | $k_{1,2}$ | $k_{1,3}$ |
| $k_{2,0}$ | $k_{2,1}$ | $k_{2,2}$ | $k_{2,3}$ |
| $k_{3,0}$ | $k_{3,1}$ | $k_{3,2}$ | $k_{3,3}$ |

$\longrightarrow k[0], k[1], k[2], k[3]$

**Algorithm 2** KeySchedule (ou KeyExpension)

**Require:** K (clé, 128 bits)
**Ensure:** $k_0, k_1, k_2, k_3, k_4$ ▷ 5 clés pour 4 tours
  $k_0 \leftarrow K$ ▷ La KeyMaster → première sous clé
  **for** $i \leftarrow 1$ *to* 4 **do** ▷ AES à 4 tours
    $tmp \leftarrow RotWord(k_{i-1}[3])$
    $tmp \leftarrow SubWord(tmp)$
    $tmp \leftarrow tmp \oplus k_i - 1[0]$
    $k_i[[0] \leftarrow tmp \oplus Rcon(i)$
    **for** $j \leftarrow 1$ *to* 3 **do**
      $k_i[j] \leftarrow k_i[j-1] \oplus k_i - 1[j]$
    **end for**
  **end for**
  **return** $k_0, k_1, k_2, k_3, k_4$

### 2.6.1 RotWord

RotWord est une rotation simple d'octet sur une colonne. C'est une permutation circulaire vers le haut.

$$\begin{array}{|c|}\hline a_0 \\ \hline a_1 \\ \hline a_2 \\ \hline a_3 \\ \hline \end{array} \longrightarrow \begin{array}{|c|}\hline a_1 \\ \hline a_2 \\ \hline a_3 \\ \hline a_0 \\ \hline \end{array}$$

### 2.6.2 SubWord

SubWord utilise la $S$-box présentée plus haut. Chaque octet est substitué à un autre octet.

$$\begin{array}{|c|}\hline a_0 \\ \hline a_1 \\ \hline a_2 \\ \hline a_3 \\ \hline \end{array} \longrightarrow \begin{array}{|c|}\hline S\text{-box } (a_0) \\ \hline S\text{-box } (a_1) \\ \hline S\text{-box } (a_2) \\ \hline S\text{-box } (a_3) \\ \hline \end{array}$$

### 2.6.3 Rcon

Rcon est une constante pour chaque tour. Le but de cette partie est d'éliminer la symétrie dans le cadencement de clé, de faire en sorte que chaque étape du cadencement de clé soit légèrement différente, pour éviter des possibles attaques sur le cadencement de clé. Voici un tableau représentant la constante $rcon[i]$ pour le tour $i$.

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $rcon[i]$ | 01 | 02 | 04 | 08 | 10 | 20 | 40 | 80 | 1B | 36 |

Les valeurs du tableau sont des octets. Pour le cadencement de clé, qui fait des opérations par colonne, il faut donc une colonne. Pour obtenir la colonne, la première case est celle du tableau ci-dessus. Les cases suivantes sont à zéro.
Exemple : pour le tour 2, la colonne obtenue est

| 02 |
|---|
| 00 |
| 00 |
| 00 |

# 3 L'attaque par saturation

## 3.1 Historique

Cette attaque a été découverte par *Lars Knudsen* et *David Wagner* en 2002 [1], et porte sur une version simplifiée de l'AES, une version à 4 tours. Cette attaque existe pour les versions allant jusqu'à 7 tours. Dans le cadre de ce projet, nous présentons l'attaque sur la version 4 tours.

Comme dit dans l'introduction, le but de cette attaque est de retrouver la clé de chiffrement qui a été utilisée pour chiffrer un ensemble de messages clairs. Il s'agit d'une attaque à clair choisi car l'attaquant doit être en mesure de chiffrer les messages qu'il souhaite afin de mener à bien son attaque. Nous allons voir, dans les sections qui suivent, comment l'attaquant va choisir ses messages clairs, et comment il va se servir des couples clairs-chiffrés obtenus pour mener l'attaque.

## 3.2 Description des propriétés

Dans un premier temps, rappelons qu'un message est représenté sous la forme d'une matrice de $4 \times 4$ octets, soit 16 octets. Pour mener son attaque, l'attaquant va choisir plusieurs clairs et nous nous intéressons aux 16 octets de chacune de ces matrices. Voici la notation que nous allons utiliser :

$$\mathcal{M}_j^i = \text{Octet } j \text{ de la matrice } i.$$
$$i \in \mathbb{N}^* \text{ et } 1 \leq j \leq 16$$

L'attaquant va choisir 256 clairs afin de retrouver certaines propriétés que nous décrirons dans les lignes qui suivent, donc $1 \leq i \leq 256$. Ces propriétés nous permettront plus tard de distinguer les octets de la dernière sous-clé. Ainsi, nous pourrons utiliser l'algorithme de cadencement de clé inverse afin de retrouver la clé maître.
Nous allons maintenant présenter les différentes propriétés des octets qui servirons lors de l'attaque :

- $\mathcal{C}$ : L'octet $j$ d'une matrice est noté $\mathcal{C}$ lorsque $\mathcal{M}_j^i = c$, $\forall i \in \{1, 2, ..., 256\}$ et $c$ étant une constante.
- $\mathcal{A}$ : L'octet $j$ d'une matrice est noté $\mathcal{A}$ lorsque $\mathcal{M}_j^1 \neq \mathcal{M}_j^2 \neq ... \neq \mathcal{M}_j^{256}$, tous les octets $j$ ont une valeur différente. Cette propriété implique le résultat suivant :

$$\bigoplus_{i=1}^{256} \mathcal{M}_j^i = 0$$

- $\mathcal{S}$ : L'octet $j$ d'une matrice est noté $\mathcal{S}$ lorsque la somme de tous les octets $j$ est prévisible, cette somme vaut 0 en général.
- ? : L'octet $j$ d'une matrice est noté ? si aucune information n'est connue à son propos.

Maintenant que nous connaissons ces propriétés, l'attaquant va pouvoir créer son **distingueur intégral**. Dans ce contexte, l'intégral de l'octet $j$ est $\bigoplus_{i=1}^{256} \mathcal{M}_j^i$.

## 3.3 Le distingueur

### 3.3.1 Qu'est-ce-qu'un distingueur et une attaque de ce type ?

Une attaque utilisant un distingueur permet à un attaquant de différencier des données chiffrées aléatoires (chiffrés reçus et/ou interceptés) et des données chiffrées de clairs connus.

### 3.3.2 Tour 1

Présentons le distingueur au fur et à mesure des tours. Dans un premier temps, l'attaquant choisi ses 256 clairs comme ceci :

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |

Cette représentation signifie que la valeur du premier octet n'est jamais la même pour les 256 matrices. Les fonctions $SubBytes$ puis $ShiftRows$ s'exécutent sur toutes les matrices et on obtient le distingueur suivant :

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |

Les propriétés restent inchangées. En effet, *SubBytes* est une fonction de substitution qui, à un octet, associe un autre octet. Il s'agit en réalité d'une bijection car chaque octet est le résultat de la substitution d'un seul et unique octet. Il en découle que les octets 1 des 256 états restent tous différents, et les autres octets restent tous égaux, mais à une nouvelle constante.

La fonction $ShiftRows$, quant à elle, ne fait que déplacer les octets de chaque lignes comme vu plus haut dans ce document. La première ligne n'étant pas affectée par cette fonction, le distingueur reste inchangé.

Vient ensuite la fonction $MixColumns$, qui est la première à changer le distingueur :

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |

La fonction $MixColumns$ a été expliquée plus haut dans ce document, mais revenons un peu dessus. Notons $A$ la matrice de $MixColumns$ spécifiée par l'AES, $B$ la première colonne du distingueur, et $C$ la colonne telle que $C = A \times B$ :

$$\begin{cases} C_0 = 2 \bullet B_0 \oplus 3 \bullet B_1 \oplus 1 \bullet B_2 \oplus 1 \bullet B_3 \\ C_1 = 1 \bullet B_0 \oplus 2 \bullet B_1 \oplus 3 \bullet B_2 \oplus 1 \bullet B_3 \\ C_2 = 1 \bullet B_0 \oplus 1 \bullet B_1 \oplus 2 \bullet B_2 \oplus 3 \bullet B_3 \\ C_3 = 3 \bullet B_0 \oplus 1 \bullet B_1 \oplus 1 \bullet B_2 \oplus 2 \bullet B_3 \end{cases}$$

Cependant, avant la fonction $MixColumns$, la colonne avait les propriétés $(\mathcal{A}, \mathcal{C}, \mathcal{C}, \mathcal{C})$.
Donc les octets numéro 1 de la colonne sont les mêmes pour tous les états, il en va de même pour les 2 autres octets. On a donc, quelque soit l'état :

$$\begin{cases} 3 \bullet B_1 \oplus 1 \bullet B_2 \oplus 1 \bullet B_3 = c_0 \\ 2 \bullet B_1 \oplus 3 \bullet B_2 \oplus 1 \bullet B_3 = c_1 \\ 1 \bullet B_1 \oplus 2 \bullet B_2 \oplus 3 \bullet B_3 = c_2 \\ 1 \bullet B_1 \oplus 1 \bullet B_2 \oplus 2 \bullet B_3 = c_3 \end{cases}$$

Avec $c_0$, $c_1$, $c_2$ et $c_3$ des constantes. On a donc :

$$\begin{cases} C_0 = 2 \bullet B_0 \oplus c_0 \\ C_1 = 1 \bullet B_0 \oplus c_1 \\ C_2 = 1 \bullet B_0 \oplus c_2 \\ C_3 = 3 \bullet B_0 \oplus c_3 \end{cases}$$

Le $B_0$ de toutes les matrices sont différents, comme explicité par la propriété $\mathcal{A}$. Donc tous les $C_i,\ i \in \{0, 1, 2, 3\}$ sont différents. La preuve :

$C_i$ est de la forme $a \bullet B_0 \oplus b$, avec $a$ et $b$ des constantes. Ce sont des fonctions affines, l'octet $C_i$ est égal à celui d'une autre matrice si et seulement si les deux octets $B_0$ sont égaux. Or, ils ne le sont pas, d'où le fait que l'on obtienne une colonne avec les propriétés $(\mathcal{A}, \mathcal{A}, \mathcal{A}, \mathcal{A})$. Le même raisonnement explique pourquoi une colonne aux propriétés $(\mathcal{C}, \mathcal{C}, \mathcal{C}, \mathcal{C})$ reste comme ceci.

Ensuite, nous avons la fonction $AddRoundKey$, qui ne change pas les propriétés du distingueur. Cette fonction XOR le $i^{eme}$ octet de toutes les matrices avec le $i^{eme}$ octet de la sous-clé correspondant au tour, nous pouvons représenter son action sur un octet comme une transformation linéaire :

$$\begin{array}{rccc} f: & \{0,1\}^{256} \times \{0,1\}^{256} & \longrightarrow & \{0,1\}^{256} \\ & (x, OctetCle) & \longmapsto & x \oplus OctetCle \end{array}$$

Si tous les octets sont différents (propriété $\mathcal{A}$), alors ils le resteront. Si tous les octets sont égaux (propriété $\mathcal{C}$), ils le resteront, mais seront égaux à une nouvelle constante.

### 3.3.3 Tour 2

Vient ensuite le second tour. *SubBytes* ne change pas les propriétés pour la même raison qu'au tour 1, tandis que $ShiftRows$ va déplacer les propriétés d'un simple décalage par ligne, comme ceci :

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{A}$ |
| $\mathcal{C}$ | $\mathcal{C}$ | $\mathcal{A}$ | $\mathcal{C}$ |
| $\mathcal{C}$ | $\mathcal{A}$ | $\mathcal{C}$ | $\mathcal{C}$ |

Le $MixColumns$ du second tour transforme la matrice de telle sorte que les 16 octets respectent la propriété $\mathcal{A}$. Comme expliqué pour le tour 1, si une colonne comporte exactement une occurrence de la propriété $\mathcal{A}$, alors toute la colonne respectera cette propriété après le $MixColumns$. Comme nous pouvons le remarquer, les 4 colonnes possèdent exactement une occurrence de la propriété $\mathcal{A}$, ce qui nous donne :

| | | | |
|---|---|---|---|
| $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |
| $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ | $\mathcal{A}$ |

Une fois de plus, la fonction $AddRoundKey$ ne change aucune propriété.

### 3.3.4 Tour 3

Lors du troisième tour comme pour les précédents, la fonction *SubBytes* ne change pas les propriétés. $ShiftRows$ effectue les rotations de lignes, mais comme tous les octets ont la propriété $\mathcal{A}$, cette fonction ne change rien au distingueur. Les 16 octets du distingueur gardent donc la propriété $\mathcal{A}$. Suite à cela, la fonction $MixColumns$ s'opère, et va changer la propriété des 16 octets car en effet, toutes les colonnes possèdent les propriétés $(\mathcal{A}, \mathcal{A}, \mathcal{A}, \mathcal{A})$. Le résultat est le suivant :

| | | | |
|---|---|---|---|
| $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ |
| $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ |
| $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ |
| $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ | $\mathcal{S}$ |

Ici, $\mathcal{S} = 0$. Expliquons cette transformation. Nous avons toujours ce système, qui représente la transformation d'une colonne :

$$\begin{cases} C_0 = 2 \bullet B_0 \oplus 3 \bullet B_1 \oplus 1 \bullet B_2 \oplus 1 \bullet B_3 \\ C_1 = 1 \bullet B_0 \oplus 2 \bullet B_1 \oplus 3 \bullet B_2 \oplus 1 \bullet B_3 \\ C_2 = 1 \bullet B_0 \oplus 1 \bullet B_1 \oplus 2 \bullet B_2 \oplus 3 \bullet B_3 \\ C_3 = 3 \bullet B_0 \oplus 1 \bullet B_1 \oplus 1 \bullet B_2 \oplus 2 \bullet B_3 \end{cases}$$

Sauf qu'ici, nous ne pouvons pas appliquer le même raisonnement que pour les deux tours précédents. Cependant, nous avons $\bigoplus_{i=1}^{256} C_j^i = 0,\ j = 0, 1, 2, 3$

Ceci s'explique mathématiquement :

$$\begin{aligned} \bigoplus_{i=1}^{256} C_j^i &= \bigoplus_{i=1}^{256} (2B_0^i \oplus 3B_1^i \oplus B_2^i \oplus B_3^i) \\ &= \begin{array}{llllllll} (2B_0^1 & \oplus & 2B_0^2 & \oplus & \ldots & \oplus & 2B_0^{256}) & \oplus \\ (3B_1^1 & \oplus & 3B_1^2 & \oplus & \ldots & \oplus & 3B_1^{256}) & \oplus \\ (B_2^1 & \oplus & B_2^2 & \oplus & \ldots & \oplus & B_2^{256}) & \oplus \\ (B_2^1 & \oplus & B_2^2 & \oplus & \ldots & \oplus & B_2^{256}) & \end{array} \quad (*) \\ &= \begin{array}{llllllll} 2 \bullet (B_0^1 & \oplus & B_0^2 & \oplus & \ldots & \oplus & B_0^{256}) & \oplus \\ 3 \bullet (B_1^1 & \oplus & B_1^2 & \oplus & \ldots & \oplus & B_1^{256}) & \oplus \\ 1 \bullet (B_2^1 & \oplus & B_2^2 & \oplus & \ldots & \oplus & B_2^{256}) & \oplus \\ 1 \bullet (B_2^1 & \oplus & B_2^2 & \oplus & \ldots & \oplus & B_2^{256}) & \end{array} \quad (**) \end{aligned}$$

(*) : Associativité et commutativité du XOR.

(**) : Factorisation.

Or, on sait que $B_0$, $B_1$, $B_2$ et $B_3$ ont la propriété $\mathcal{A}$, donc :

$$(B_j^1 \oplus B_j^2 \oplus ... \oplus B_j^{256}) = 0,\ j = 0, 1, 2, 3$$

Le calcul devient donc :

$$\bigoplus_{i=1}^{256} C_j^i = 2 \bullet 0 \oplus 3 \bullet 0 \oplus 1 \bullet 0 \oplus 1 \bullet 0 = 0$$

Ceci nous explique les propriétés $\mathcal{S}$, le résultat prévisible dans notre cas, qui est **zéro**
Comme pour les tours précédents, $AddRoundKey$ ne change aucune propriété.

### 3.3.5 Tour 4

Le quatrième tour se fait sans la fonction $MixColumns$, et la fonction $SubBytes$ casse les propriétés que nous avions jusqu'ici. En effet, il devient impossible, après la substitution, de prédire le résultat de l'intégral. On obtient le distingueur suivant :

| | | | |
|---|---|---|---|
| ? | ? | ? | ? |
| ? | ? | ? | ? |
| ? | ? | ? | ? |
| ? | ? | ? | ? |

## 3.4 Retrouver la clé de chiffrement

Afin de retrouver la clé de chiffrement utilisée pour les 256 messages clairs, l'attaquant doit remonter partiellement les étapes de chiffrement. Il peut retrouver la valeur de la dernière sous-clé octet par octet. Concentrons-nous d'abord sur le premier octet de la dernière sous-clé, notons cet octet $\oslash_0$ . L'attaquant va tester les 256 valeurs possibles de $\oslash_0$, et remonter le calcul avec chacune de ces valeurs : pour un test, il XOR le premier octet des chiffrés avec la valeur test de $\oslash_0$, ce qui donne un nouvel octet, que nous notons $\oslash_1$. Il effectue cette opération avec les 256 chiffrés $(C_0, C_1, ..., C_{255})$.

Pour chaque nouvel octet, il effectue la fonction $ShiftRows^{-1}$ puis $SubBytes^{-1}$. L'attaquant obtient alors 256 nouveaux octets, et si le XOR de ces 256 octets donne 0, alors il ajoute cette valeur test $\oslash_0$ dans la liste des octets candidats de la sous-clé (liste concernant le premier octet car nous nous sommes concentré dessus), car la propriété $\mathcal{S}$ du distingueur à la fin du troisième tour est respectée. Il répète l'ensemble de ces opérations pour les 15 autres octets de la sous-clé, afin d'obtenir une liste de candidat concernant tous les octets.

Voyons maintenant la complexité de ces opérations : $2^8$ valeurs test pour chaque octet et il y a au total $2^4$ octets (matrice $4 \times 4$) pour $2^8$ chiffrés (l'attaquant dispose de 256 couples clair-chiffré). $2^8 \times 2^4 \times 2^8 = 2^{20}$.

Au bout de ces calculs, l'attaquant dispose d'une petite liste de valeurs possibles, pour chacun des 16 octets de la sous-clé, donc 16 listes. Une possibilité serait de tester récursivement toutes les combinaisons possibles de ces valeurs candidates (chaque octet de la première liste, avec chaque octet de la second liste, ...). Pour chaque sous-clé à tester, l'attaquant effectue l'inverse du $KeySchedule$ afin de trouver la clé maître correspondante, puis chiffre les 256 clairs dont il dispose. Si tous les chiffrés correspondent aux clairs comme indiqué par les couples, alors cette clé maître est la bonne, et l'attaque est réussie. Sinon, il passe au test suivant.

Une autre solution serait de créer 256 autres couples clair-chiffré avec des constantes différentes (au niveau des propriétés $\mathcal{C}$), et d'effectuer l'intersection des 16 listes du premier ensemble de couples avec les 16 du deuxième ensemble de couples. Ceci permet d'obtenir une seule valeur possible de la dernière sous-clé, l'attaquant n'a alors plus qu'à inverser le $KeySchedule$ pour retrouver la clé maître.

Nous utilisons trois ensembles de 256 couples clair-chiffré afin de rendre l'intercetion efficace. Cela signifie que nous devons effectuer à trois reprises les opérations consistant à déterminer les listes d'octets candidats. Voici la compléxité de cette solution : $3 \times 2^{20} \approx 2^{21.58}$ pour les listes, ce qui reste trivial pour les ordinateurs modernes. L'intercetion des listes est un algorithme en temps linéaire, négligeable.

### 3.5 Statistiques de l'attaque

Lors de l'attaque, nous créons des listes d'octets potentiels. Lorsque l'on lance le programme avec différentes clés et que nous faisons la moyenne du nombre d'octets potentiels de chaque liste, nous obtenons des résultats très proches de 2. Il y a donc en moyenne deux candidats pour chacun des 16 octets de la dernière sous-clé. Nous pouvons maintenant approcher la complexité de l'attaque utilisant la récursivité : $\underbrace{2 \times 2 \times ... \times 2}_{16 \text{ fois}} = 2^{16}$ sous-clés à tester en moyenne.

## 4 Choix d'implémentation - Code

### 4.1 Type de variable utilisé

Afin de manipuler les blocs d'états de l'AES, nous avions besoin de représenter ce qu'est un octet. Un octet faisant 8 bits, une valeur d'octet peut prendre une valeur entière comprise entre 0 et 255 (0x00 et 0xff en hexadécimal).
Nous aurions pu choisir le type *char* mais pour des raisons de conversion, le type défini dans le standard C99, $uint8\_t$ nous permet de plus facilement représenter l'octet. Pour plus de lisibilité, ce type a été renommé *byte*.

### 4.2 Tables S

La table S, définie dans $headers/S\_box.h$, est donc un tableau de *byte* ayant une taille de 16 par 16.

### 4.3 Interface en ligne de commande

Notre programme n'ayant pas d'interface graphique, nous interagissons avec l'utilisateur avec l'entrée standard.
Les commandes disponibles lors de l'exécution sont vérifiées et exécutées à l'aide des fonctions définies dans $headers/CLI.h$.

### 4.4 Gestion d'erreurs

Des erreurs peuvent subvenir lors de l'exécution par une mauvaise entrée utilisateur ou encore l'allocation de mémoire. Ces erreurs sont ainsi gérées par les fonctions définies dans $headers/Errors.h$.

## 5 Sources

- Fonctionnement de l'AES : `https://www.youtube.com/watch?v=lnKPoWZnNNM`
- Implémentation de l'AES : `https://www.davidwong.fr/blockbreakers/aes.html`
- MixColumns : `https://en.wikipedia.org/wiki/Rijndael_MixColumns`
- S-box : `https://en.wikipedia.org/wiki/Rijndael_S-box`

## References

[1] Lars Knudsen, David Wagner (2002) *Integral Cryptanalysis*, p. 112-117.