Travaux pratiques de l'UE « Cryptographie »

Liste des exercices

Version du 14 janvier 2020


Un exercice à faire chez soi *avant* le TD/TP.


Un exercice à faire absolument.


Un exercice à finir chez soi éventuellement.


Un exercice à rendre.


Extrait des annales d'examens et de projets.


Un indice pour ne pas se tromper.


Une question facile.


Une question pour réfléchir un peu.

L'AES est un système cryptographique symétrique par blocs recommandé par de nombreuses agences de sécurité. Dans ce protocole, chaque bloc comporte 128 bits, c'est-à-dire 16 octets, vus comme une matrice 4×4, appelée *State* (figure 18).

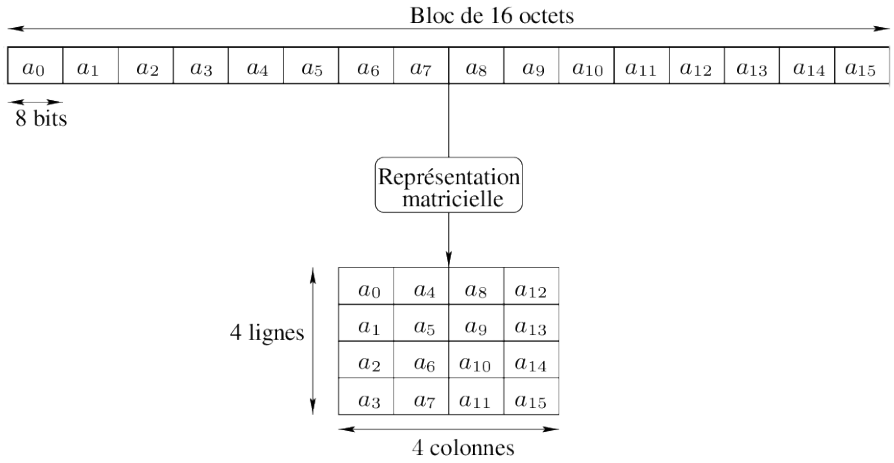

FIGURE 18 – Placement des octets d'un bloc dans la matrice 4×4

L'AES utilise, au choix, des clefs secrètes de longueur 128, 192 ou 256 bits. On note N_k la longueur de la clé secrète K en nombre de mots (de 32 bits) : N_k vaut donc 4, 6 ou 8. Le chiffrement d'un bloc consiste à opérer une série de transformations appelées *rondes* : le nombre N_r de rondes qui doivent être effectuées lors du chiffrement (ou du déchiffrement) dépend de la longueur de la clef secrète, c'est-à-dire N_k, de la manière suivante : $N_k = 4 \rightsquigarrow N_r = 10$; $N_k = 6 \rightsquigarrow N_r = 12$; $N_k = 8 \rightsquigarrow N_r = 14$.

AES_ENCRYPT$(State, K)$
1 **KeyExpansion**$(K, RoundKeys)$;
2 **AddRoundKey**$(State, RoundKeys[0])$;
3 **for** $i \leftarrow 1$ **to** $N_r - 1$
4 **do** // Boucle de $N_r - 1$ rondes constituées chacune de 4 opérations
5 **SubBytes**$(State)$;
6 **ShiftRows**$(State)$;
7 **MixColumns**$(State)$;
8 **AddRoundKey**$(State, RoundKeys[i])$;
9 **SubBytes**$(State)$;
10 **ShiftRows**$(State)$;
11 **AddRoundKey**$(State, RoundKeys[N_r])$;
12 **return**

FIGURE 19 – Algorithme de chiffrement AES d'un bloc

Le chiffrement d'un bloc, décrit sur la figure 19, fait intervenir $N_r + 1$ *clefs de ronde*. Chaque clef de ronde est également une matrice 4×4 d'octets ; elle est « ajoutée » à la matrice *State* par l'opération `AddRoundKey`. Les clefs de ronde sont obtenues en extrayant des morceaux de la clef longue W elle-même obtenue à partir de la clef courte K, selon le procédé *d'extension* `KeyExpansion` *de la clef de chiffrement* implémenté au TP précédent : celui-ci consiste, en fait, à fabriquer les clefs de rondes en nombre suffisant. La clef longue W est la concaténation des clefs de rondes : puisqu'il faut $N_r + 1$ clefs de rondes et que chacune d'entre elles comporte 16 octets, la clef longue W est formée de $16 \times (N_r + 1)$ octets.

Le chiffrement d'un bloc opère un certain nombre de transformations sur le bloc *State* ; le bloc chiffré est simplement le résultat de ces transformations successives. Outre l'addition de clefs de ronde (fig. 20), le chiffrement s'appuie sur trois autres opérations : `SubBytes`, `ShiftRows` et `MixColumns`, décrites respectivement sur les figures 21, 22 et 23.

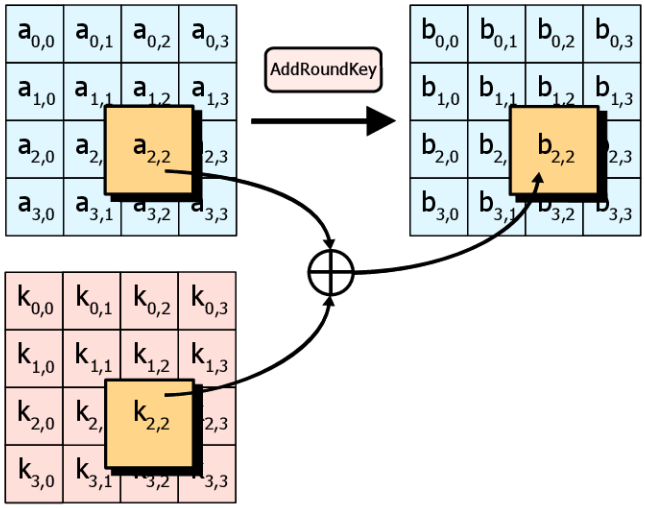

FIGURE 20 – AddRoundKey

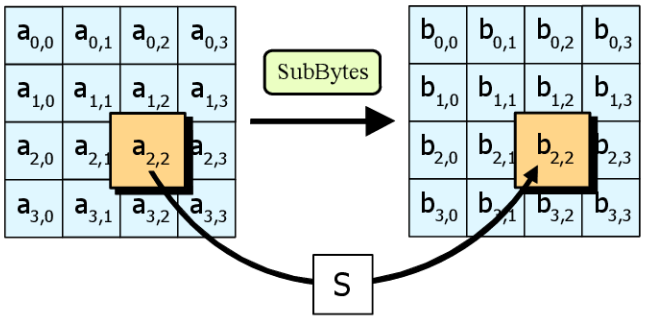

FIGURE 21 – SubBytes

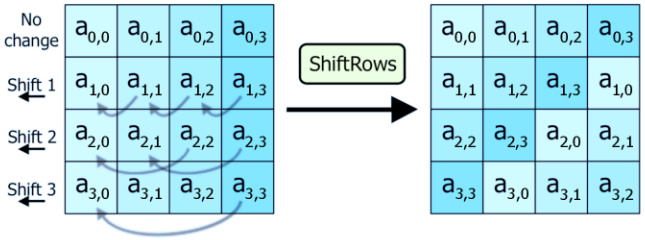

FIGURE 22 – ShiftRows

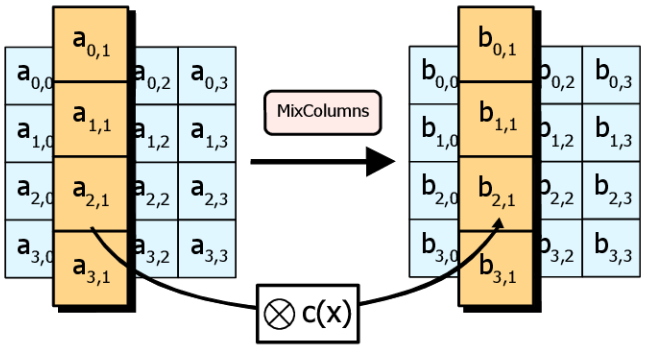

FIGURE 23 – MixColumns

Opérations effectuées au cours d'une ronde

Le chiffrement d'un bloc commence par l'addition, notée **AddRoundKey**, de la première clef de ronde `RoundKeys`[0] à la matrice *State* (fig. 20). Il s'agit simplement d'opérer un XOR octet par octet entre les 16 octets du bloc et les 16 premiers octets de la clef longue W. Puis $N_r - 1$ rondes sont effectuées ; chacune est constituée de quatre étapes :

① **SubBytes** : il s'agit d'une opération de *substitution* qui remplace chaque octet de *State* par un autre octet déterminé par une table particulière (la « SBox »), déjà utilisée pour la fabrique des clefs de rondes (fig. 14 du TP B). Il s'agit donc d'une opération semblable à **SubWord** appliquée non pas à un vecteur colonne mais à toute la matrice *State*.

② **ShiftRows** est une étape de *transposition* dans laquelle les éléments du bloc sont décalés de façon circulaire : *c'est l'opération la plus simple de la ronde* : les éléments de la première ligne ne bougent pas ; ceux de la seconde ligne se déplacent d'une case vers la gauche ; ceux de la troisième ligne se déplacent de deux cases vers la gauche ; enfin, les éléments de la quatrième ligne se déplacent de trois cases vers la gauche (c'est-à-dire d'une case vers la droite).

③ L'opération **MixColumns** effectue un produit matriciel entre le bloc et une matrice déterminée ; cela revient à calculer le vecteur colonne correspondant à chaque vecteur colonne de 4 octets du bloc *State*. L'opération **MixColumns** consiste ainsi à remplacer dans le bloc *State* chaque colonne $(a_0, a_1, a_2, a_3)^\top$ de 4 octets par la colonne $(b_0, b_1, b_2, b_3)^\top$ obtenue par le produit matriciel ci-dessous :

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

c'est-à-dire :

— $b_0 = \texttt{0x02} \times a_0 + \texttt{0x03} \times a_1 + \texttt{0x01} \times a_2 + \texttt{0x01} \times a_3$

— $b_1 = \texttt{0x01} \times a_0 + \texttt{0x02} \times a_1 + \texttt{0x03} \times a_2 + \texttt{0x01} \times a_3$

— $b_2 = \texttt{0x01} \times a_0 + \texttt{0x01} \times a_1 + \texttt{0x02} \times a_2 + \texttt{0x03} \times a_3$

— $b_3 = \texttt{0x03} \times a_0 + \texttt{0x01} \times a_1 + \texttt{0x01} \times a_2 + \texttt{0x02} \times a_3$

L'addition de deux octets consiste, à nouveau, à opérer un XOR. Pour la multiplication de deux octets, vous ferez appel à la fonction `gmul` donnée dans le programme à compléter [1].

④ Puis l'opération **AddRoundKey** ajoute à chaque octet du bloc *State* courant l'octet correspondant de la clef de ronde choisie (fig. 20).

Enfin, une ronde finale est appliquée : elle correspond à une ronde normale dans laquelle néanmoins l'étape **MixColumns** est omise. Il faut donc bien disposer de $N_r + 1$ clefs de ronde.

Objectif de cette séance de TP

Il s'agit aujourd'hui tout d'abord d'écrire un programme AES qui calcule et affiche le chiffré du bloc nul (c'est-à-dire formé de 16 octets nuls), en utilisant la clef nulle, formée de 16 octets nuls (Exercice C.1). Vous ajouterez ensuite éventuellement au programme le calcul et l'affichage du *déchiffrement* du bloc chiffré obtenu (Exercice C.2). À partir de là, vous programmerez le chiffrement d'un fichier complet.

Le squelette du programme à compléter (en C ou en Java) est disponible dans l'archive `AES.zip` (Figures 25 et 26) ; il s'appuie sur la définition de la fontion `gmul` qui permet de multiplier deux octets et l'initialisation de plusieurs tableaux, déclarés globalement :

— `uchar K[32]` et `uchar W[240]` contiennent respectivement la clef courte (qui ne contient que des 0) et la clef étendue correspondante, d'où seront extraites les clefs de rondes.

— `uchar State[16]` est un tableau de 16 octets qui représente l'état de la matrice *State* en cours de chiffrement (ou de déchiffrement). Initialement ce tableau ne contient que des 0, puisque l'objectif est de chiffrer le bloc nul.

— `uchar SBox[256]` contient la table de substitution à utiliser pour la tranformation **SubBytes**.

1. Clairement, l'octet nul `0x00` est l'élément neutre pour l'addition : $\texttt{0x00} \oplus a = a \oplus \texttt{0x00} = a$ pour chaque octet a. D'autre part, `0x01` apparaît comme l'élément neutre de la multiplication : $\texttt{0x01} \times a = a \times \texttt{0x01} = a$ pour chaque octet a. De plus, l'octet nul `0x00` est *absorbant* : $\texttt{0x00} \times a = a \times \texttt{0x00} = \texttt{0x00}$. Enfin, tout octet a possède un inverse b, c'est-à-dire que $a \times b = b \times a = \texttt{0x01}$. Munis de cette multiplication et de cette addition, les octets forment ce que les mathématiciens appellent un *corps*, souvent noté $\mathbb{F}_{256}$ ou GF(2^8).

Exercice C.1 Chiffrement d'un bloc Il s'agit de compléter l'un des squelettes de programme donné sur les figures 25 et 26 afin d'effectuer le chiffrement du bloc formé de 16 octets nuls à l'aide de la clef courte formée de 16 octets nuls. Le résultat attendu correpond à l'exécution de la commande suivante :

```
$ echo -e -n "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"|openssl enc -aes-128-ecb -K 0 -nopad|xxd
00000000: 66e9 4bd4 ef8a 2c3b 884c fa59 ca34 2b2e  f.K...,;.L.Y.4+.
$
```

Question 1. Écrire une fonction **`void SubBytes()`** qui remplace chaque octet **`x`** du tableau **`State`** par **`SBox[x]`**. Testez cette fonction en l'appliquant sur le bloc **`State`** dans lequel chaque ligne i contient 4 octets égaux à i, en affichant le résultat et en le comparant au résultat indiqué sur la figure 27.

Question 2. Écrire *de manière concise* une fonction **`void ShiftRows()`** qui déplace les octets du tableau **`State`** comme indiqué plus haut. Testez cette fonction afin de visualiser le résultat comme sur la figure 27.

Question 3. Écrire une fonction **`void MixColumns()`** qui implémente la tranformation **MixColumns** de *State* définie plus haut. Testez votre code à l'aide du calcul indiqué sur la figure 24 en comparant le résultat obtenu avec la figure 27.

Question 4. Ajouter une fonction **`void AddRoundKey(int r)`** qui effectue ajoute au bloc **`State`** la r-ième clef de ronde, stockée dans le tableau **`W`**. Le résultat sera placé dans **`State`**. L'addition consiste en un simple XOR, octet par octet. Testez cette fonction en affichant la seconde clef de ronde puis le résultat de l'addition de cette clef sur le bloc *State* ne contenant que des **`0xFF`**. Vérifiez le résultat en le comparant à celui indiqué sur la figure 27.

Question 5. Calculer et afficher le chiffré du bloc nul (ne contenant que des **`0x00`**) par la clef nulle de longueur 16 octets. Comparez avec le résultat indiqué sur la figure 27.

$$\begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \times \begin{pmatrix} \texttt{0x0E} & \texttt{0x0B} & \texttt{0x0D} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0E} & \texttt{0x0B} & \texttt{0x0D} \\ \texttt{0x0D} & \texttt{0x09} & \texttt{0x0E} & \texttt{0x0B} \\ \texttt{0x0B} & \texttt{0x0D} & \texttt{0x09} & \texttt{0x0E} \end{pmatrix} = \begin{pmatrix} \texttt{0x01} & \texttt{0x00} & \texttt{0x00} & \texttt{0x00} \\ \texttt{0x00} & \texttt{0x01} & \texttt{0x00} & \texttt{0x00} \\ \texttt{0x00} & \texttt{0x00} & \texttt{0x01} & \texttt{0x00} \\ \texttt{0x00} & \texttt{0x00} & \texttt{0x00} & \texttt{0x01} \end{pmatrix}$$

FIGURE 24 – Calcul utile pour tester **`MixColumns`** (et l'inverser)

Exercice C.2 Déchiffrement d'un bloc Pour déchiffrer un bloc, il suffit d'opérer les opérations inverses du chiffrement, une par une, mais dans l'ordre inverse du chiffrement.

Question 1. Écrire une fonction **`Inv_SubBytes()`** qui inverse la modification réalisée par **`SubBytes()`**. Vous pourrez pour cela créer un tableau qui matérialise la substitution à utiliser pour opérer l'opération inverse de **SubBytes**.

Question 2. Testez la fonction **`Inv_SubBytes()`**.

Question 3. L'égalité matricielle de la figure 24 fournit la matrice inverse de celle utilisée pour l'opération **MixColumns**. Écrire une fonction **`void Inv_MixColumns()`** qui rétablit la valeur de **`State`**, c'est-à-dire qui effectue l'opération inverse de **`MixColumns()`**.

Question 4. Testez la fonction **`void Inv_MixColumns()`**.

Question 5. Ajouter une fonction **`void Inv_ShiftRows()`** qui déplace les octets du tableau **`State`** dans le sens inverse, c'est-à-dire qui annule les modifications réalisées par **`ShiftRows()`**.

Question 6. Testez la fonction **`void Inv_ShiftRows()`**.

Question 7. Compléter le programme `aes.c` afin qu'il affiche également le déchiffrement du bloc précédemment chiffré. Vous devez naturellement obtenir le bloc initial en guise de résultat final. Testez-le.

```c
typedef unsigned char uchar;

/* La clef courte K utilisée est la clef formée de 16 octets nuls */
int longueur_de_la_clef = 16 ;
uchar K[16] = {0x00, 0x00, 0x00, 0x00, ..., 0x00, 0x00, 0x00, 0x00} ;

/* Résultat du TP précédent */
int Nr = 10, Nk = 4;
int longueur_de_la_clef_etendue = 176;          // C'est 16 * (Nr + 1)
uchar W[176] = {0x00, 0x00, 0x00, 0x00, ..., 0x6F, 0x8F, 0x18, 0x8E } ;

/* Le bloc à chiffrer aujourd'hui: 16 octets nuls */
uchar State[16] = {0x00, 0x00, 0x00, 0x00, ..., 0x00, 0x00, 0x00, 0x00} ;

/* Programme principal */
void chiffrer(void);
void afficher_le_bloc(uchar *M);

int main(void)
{
  printf("Le bloc \"State\" en entrée vaut : \n");
  afficher_le_bloc(State);
  chiffrer();
  printf("Le bloc \"State\" en sortie vaut : \n");
  afficher_le_bloc(State);
  exit(EXIT_SUCCESS);
}

void afficher_le_bloc(uchar *M)
{
  for (int i=0; i<4; i++) { // Lignes 0 à 3
    printf("          ");
    for (int j=0; j<4; j++) { // Colonnes 0 à 3
      printf ("%02X ", M[4*j+i]); }
    printf("\n");
  }
}

void chiffrer(void)
{
  AddRoundKey(0);
  for (int i = 1; i < Nr; i++) {
    SubBytes();
    ShiftRows();
    MixColumns();
    AddRoundKey(i);
  }
  SubBytes();
  ShiftRows();
  AddRoundKey(Nr);
}

/* Table de substitution déjà utilisée lors du TP précédent */
uchar SBox[256] = {0x63, 0x7C, 0x77, 0x7B, ..., 0xB0, 0x54, 0xBB, 0x16};

/* Fonction mystérieuse qui calcule le produit de deux octets */
uchar gmul(uchar a, uchar b) { ... }

/* Partie à compléter pour ce TP */
void SubBytes(void){};
void ShiftRows(void){};
void MixColumns(void){};
void AddRoundKey(int r){};
```

FIGURE 25 – Programme `aes.c` à compléter

```java
public class Aes {
  /* La clef courte K utilisée aujourd'hui est formée de 16 octets nuls */
  int longueur_de_la_clef = 16 ;
  byte K[] = { (byte) 0x00, (byte)0x00, (byte)0x00, ..., (byte)0x00 } ;

  /* Résultat du TP précédent: diversification de la clef K en une clef étendue W */
  static int Nr = 10;
  static int Nk = 4;
  int longueur_de_la_clef_etendue = 176;
  public byte W[] = { (byte)0x00, (byte)0x00, (byte)0x00, ..., (byte)0x18, (byte)0x8E};

  /* Le bloc à chiffrer aujourd'hui: 16 octets nuls */
  public byte State[] = { (byte)0x00, (byte)0x00, (byte)0x00, ..., (byte)0x00 };

  /* Programme principal */
  public static void main(String args[]) {
    Aes aes = new Aes() ;
    System.out.println("Le bloc \"State\" en entrée vaut : ") ;
    aes.afficher_le_bloc(aes.State) ;
    aes.chiffrer() ;
    System.out.println("Le bloc \"State\" en sortie vaut : ") ;
    aes.afficher_le_bloc(aes.State) ;
  }

  public void afficher_le_bloc(byte M[]) {
    for (int i=0; i<4; i++) { // Lignes 0 à 3
      System.out.print("          ");
      for (int j=0; j<4; j++) { // Colonnes 0 à 3
        System.out.print(String.format("%02X ", M[4*j+i]));
      }
      System.out.println();
    }
  }

  public void chiffrer(){
    AddRoundKey(0);
    for (int i = 1; i < Nr; i++) {
      SubBytes();
      ShiftRows();
      MixColumns();
      AddRoundKey(i);
    }
    SubBytes();
    ShiftRows();
    AddRoundKey(Nr);
  }

  /* Table de substitution déjà utilisée lors du TP précédent */
  public byte[] SBox = { (byte)0x63, (byte)0x7C, ..., (byte)0xBB, (byte)0x16};

  /* Fonction mystérieuse qui calcule le produit de deux octets */
  byte gmul(byte a1, byte b1) { ... }

  /* Partie à compléter pour ce TP */
  public void SubBytes(){}
  public void ShiftRows(){}
  public void MixColumns(){}
  public void AddRoundKey(int r){}
}
```

FIGURE 26 – Programme `Aes.java` à compléter

```
$ make
gcc aes.c -o aes -lm -std=c99
$ ./aes
Test de SubBytes():
Le bloc "State" en entrée vaut :
          00 00 00 00
          01 01 01 01
          02 02 02 02
          03 03 03 03
Le bloc "State" en sortie vaut :
          63 63 63 63
          7C 7C 7C 7C
          77 77 77 77
          7B 7B 7B 7B

Test de ShiftRows():
Le bloc "State" en entrée vaut :
          A0 A1 A2 A3
          B0 B1 B2 B3
          C0 C1 C2 C3
          D0 D1 D2 D3
Le bloc "State" en sortie vaut :
          A0 A1 A2 A3
          B1 B2 B3 B0
          C2 C3 C0 C1
          D3 D0 D1 D2

Test de MixColumns():
Le bloc "State" en entrée vaut :
          0E 0B 0D 09
          09 0E 0B 0D
          0D 09 0E 0B
          0B 0D 09 0E
Le bloc "State" en sortie vaut :
          01 00 00 00
          00 01 00 00
          00 00 01 00
          00 00 00 01

Test de AddRoundKey():
La clef de ronde 1 vaut :
          62 62 62 62
          63 63 63 63
          63 63 63 63
          63 63 63 63
Le bloc "State" en entrée vaut :
          FF FF FF FF
          FF FF FF FF
          FF FF FF FF
          FF FF FF FF
Le bloc "State" en sortie vaut :
          9D 9D 9D 9D
          9C 9C 9C 9C
          9C 9C 9C 9C
          9C 9C 9C 9C

Test final: chiffrement du bloc nul:
Le bloc "State" en entrée vaut :
          00 00 00 00
          00 00 00 00
          00 00 00 00
          00 00 00 00
Le bloc "State" en sortie vaut :
          66 EF 88 CA
          E9 8A 4C 34
          4B 2C FA 2B
          D4 3B 59 2E
```

FIGURE 27 – Ensemble de tests proposés pour les fonctions du programme AES

Exercice C.3 Bourrage standard selon le PKCS#5 Certains algorithmes de chiffrement, tels que l'AES, supposent que la longueur du fichier à chiffrer est un multiple de k octets, pour une valeur de k déterminée. Pour de tels algorithmes, la phase de « bourrage » (ou « padding » en anglais) consiste à compléter le fichier à l'aide d'octets supplémentaires. Plus précisément, si le tableau d'octets à chiffrer contient l octets, alors il est étendu à l'aide de $k - (l \bmod k)$ octets supplémentaires. Le standard PKCS#5 (et PKCS#7) stipule qu'il faut en fait ajouter $k - (l \bmod k)$ octets tous égaux et dont la valeur est précisément $k - (l \bmod k)$:

```
In other words, the input is padded at the trailing end with one of the
following strings:
                    01 -- if l mod k = k-1
                 02 02 -- if l mod k = k-2
                    .
                    .
                    .
          k k ... k k -- if l mod k = 0
```

En particulier, si la taille du fichier d'origine est déjà un multiple de k, alors k octets seront tout de même ajoutés au fichier d'origine et leur valeur commune [2] sera k.

Il s'agit à présent de chiffrer par l'AES le fichier `butokuden.jpg` de l'archive `HMAC.zip`. Ce fichier doit tout d'abord subir cette opération de bourrage avant que ne s'applique le chiffrement par bloc. Puisqu'il contient 467 796 octets, il faudra lui adjoindre 12 octets [3] ; de plus, les 12 octets ajoutés au fichier d'origine seront tous égaux à **`0x0C`** (c'est-à-dire 12 en décimal).

Question 1. Écrivez un programme `pkcs5` qui prend en paramètre d'entrée le nom d'un fichier et qui fabrique le fichier associé selon la technique du bourrage PKCS#5 décrite ci-dessus. Votre programme devra naturellement fonctionner quelle que soit la taille du fichier d'origine.

Question 2. Exécutez votre programme sur le fichier `butokuden.jpg` afin d'obtenir le fichier associé, conservé sous le nom `pkcs5-butokuden.jpg`; puis vérifiez que ce dernier est correct à l'aide de la commande **`od -tu1`**.

Question 3. Écrivez un programme `inv_pkcs5` qui prend en paramètre d'entrée le nom d'un fichier et qui, si c'est possible, retire de ce fichier le bourrage PKCS#5 qu'il contient et sinon affiche un message d'erreur.

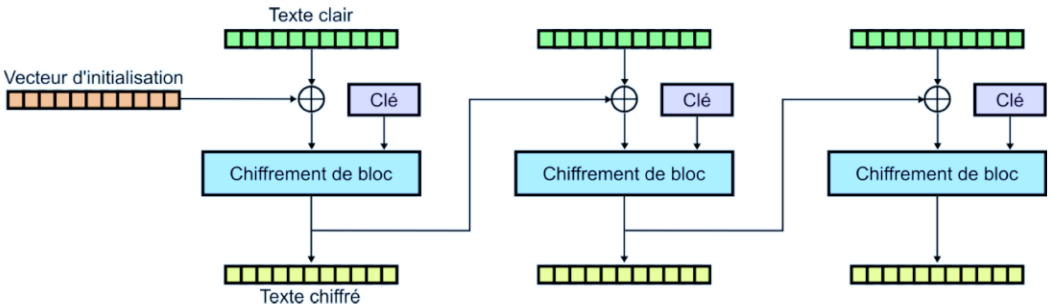

FIGURE 28 – Schéma du mode opératoire CBC

Exercice C.4 Mode opératoire CBC Il s'agit maintenant de chiffrer par l'AES le fichier `butokuden.jpg` de l'archive `HMAC.zip` selon le mode opératoire CBC (Cipher Block Chaining) décrit sur la figure 28. Comme dans tous les modes opératoires, le fichier d'origine est découpé en une suite de blocs $m_1, m_2, ..., m_n$ (de 16 octets chacun, car il s'agit d'AES) ; le fichier produit sera alors formé par les blocs $c_1, c_2, ..., c_n$ définis comme suit :

1. $c_1 = E_K(m_1 \oplus \text{IV})$ où IV est le *vecteur d'initialisation* formé de 16 octets ;
2. Pour chaque i compris entre 2 et n : $c_i = E_K(m_i \oplus c_{i-1})$.

2. Cette règle n'est évidemment applicable que pour des blocs dont la taille k du bloc est inférieure à 255, ce qui est le cas pour l'AES puisqu'alors $k = 16$.

3. car les blocs de l'AES font 16 octets et le nombre 467 808 est bien un multiple de 16

Dans ces formules, la fonction $E_K(b)$ représente le chiffrement AES d'un bloc b par la clef K et le symbole $\oplus$ désigne le XOR bit-à-bit. Ainsi, le premier bloc chiffré c_1 est obtenu en appliquant d'abord un XOR entre les 16 premiers octets du fichier d'origine et les 16 octets du vecteur d'initialisation ; les 16 octets ainsi obtenus seront chiffrés à l'aide de la clef, c'est-à-dire en exécutant la fonction **`chiffrer()`** de l'exercice C.1. Les blocs suivants du fichier chiffré seront obtenus de manière analogue, le bloc chiffré précédent c_{i-1} jouant à chaque étape le rôle du vecteur d'initialisation.

Question 1. Modifiez le programme `aes` afin qu'il lise le fichier `butokuden.jpg` et fabrique le fichier chiffré `cbc-secret.jpg`. La clef de chiffrement employée sera ici encore la clef formée de 16 octets nuls : la clef étendue et les clefs de rondes seront donc celles obtenues au TP précédent. Le *vecteur d'initialisation (IV)* choisi sera également formé de 16 octets nuls.
La phase de bourrage PKCS#5 implémentée dans l'exercice précédent doit être utilisée, afin de travailler sur des blocs « complets ». Néanmoins il faudra éviter d'utiliser le fichier intermédiaire `pkcs5-butokuden.jpg` de l'exercice précédent ou de stocker tout le fichier en mémoire, ce qui pourrait s'avérer trop coûteux. Notez que la taille du fichier chiffré `cbc-secret.jpg` doit être égale à la taille du fichier intermédiaire `pkcs5-butokuden.jpg`, c'est-à-dire 467 808 octets.

Question 2. Dans un terminal, comparez le résumé MD5 du fichier `cbc-secret.jpg` produit au résultat obtenu par la commande suivante :

```
$ openssl enc -aes-128-cbc -K 0 -iv 0 -in butokuden.jpg > butokuden_c.jpg
$ md5sum butokuden_c.jpg
MD5 (butokuden_c.jpg) = 54a8554a31b0f7c7ba3152b3f1758378
```

Question 3. Modifiez votre programme afin de choisir de manière aléatoire un vecteur d'initialisation de 16 octets au début de chaque chiffrement. Vérifiez que le fichier `cbc-secret.jpg` obtenu est différent à chaque exécution.

Question 4. Faites ensuite en sorte de stocker le vecteur d'initialisation choisi aléatoirement au *début* du fichier `cbc-secret.jpg`. Vérifiez que la taille du fichier `cbc-secret.jpg` est maintenant supérieure de 16 octets à la taille du fichier `pkcs5-butokuden.jpg`.

Question 5. Conservez dans vos archives le fichier `cbc-secret.jpg` obtenu : vous pourrez vérifier sa validité lors d'un prochain TP à l'aide de la JCE. Vous pouvez également tester la validité du fichier `cbc-secret.jpg` à l'aide d'`openssl` en lançant les commandes suivantes :


```
$ head -c 16 cbc-secret.jpg | xxd
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
$ tail -c +17 cbc-secret.jpg | wc -c
  467808
$ tail -c +17 cbc-secret.jpg | openssl enc -aes-128-cbc -d -K 0
  -iv $(head -c 16 cbc-secret.jpg| xxd -ps) > butokuden_d.jpg
$ diff butokuden_d.jpg butokuden.jpg
$
```