*RC4 est un algorithme de chiffrement à flot conçu en 1987 par Ronald Rivest, l'un des inventeurs du RSA. Jamais breveté (même si le nom RC4 est une marque déposée), ce système de chiffrement par flot est fréquemment utilisé pour le WEP, le WPA, les protocoles de cryptage proposés dans BitTorrent, PDF, etc. Il est aussi disponible en option dans SSL, SSH, etc. Les raisons de son succès sont liées à sa grande simplicité et à sa vitesse de chiffrement. Les implémentations matérielles ou logicielles sont faciles à mettre en œuvre.*

---

Le système RC4 est un chiffrement de type VERNAM : le texte chiffré est simplement le XOR, octet-par-octet, du texte clair et d'une clef longue `W` ; cette dernière doit donc être de la même longueur que le texte clair. Pour générer le flot d'octets de la clef longue `W` à partir de la clef secrète `C`, l'algorithme RC4 s'appuie sur un *état interne* qui comprend

① un tableau `S` de 256 octets, tous différents : c'est donc une *permutation* des 256 octets possibles ;

② deux pointeurs `i` et `j`, sur 8 bits, qui servent d'index dans ce tableau.

**Phase d'initialisation de l'état interne, à l'aide de la clef courte `C`**
Au départ, le $k$-ième élément du tableau `S` contient l'octet $k$, ce qui conduit à l'état suivant :

| S = | 0x00 | 0x01 | 0x02 | ... | 0xFF |
|---|---|---|---|---|---|

Le tableau `S` est ensuite modifié selon la clef courte `C`, de taille `L` variable. Cette clef courte est vue comme un tableau d'octets : `C[0]` ... `C[L-1]`. La phase d'initialisation se poursuit en mélangeant les octets du tableau `S` selon les 256 échanges suivants :

```
j := 0
pour i de 0 à 255
    j := (j + S[i] + C[i mod L]) mod 256
    échanger(S[i], S[j])
finpour
```

Notez que chaque élément de `S` est échangé au moins une fois, puisque `i` parcourt tout le tableau. D'autre part, si la clef est trop courte ($L \leqslant 255$), alors « on la répète » (c'est le rôle du `mod L`).

**Phase de production de la clef longue `W` aussi longue que le texte clair**
Pour produire un octet de la clef longue `W`, il suffit de choisir un octet `w` particulier dans le tableau `S` ; ce choix repose sur les valeurs courantes des deux indices `i` et `j`, initialisés à 0.

1. La somme des octets pointés par `i` et `j` dans `S` est calculée *modulo 256*.
2. Le résultat de cette somme détermine la position dans `S` qui contient l'octet `w` produit.

```
w := S[(S[i] + S[j]) mod 256]
```

Cependant, avant la production de chaque octet, l'état interne est modifié selon les trois instructions suivantes :

```
i := (i + 1) mod 256
j := (j + S[i]) mod 256
échanger(S[i], S[j])
```

**Exercice B.1 Chiffrement et déchiffrement de Vernam** La production et l'affichage en hexadécimal des premiers octets de la clef longue à partir d'une clef courte est codée en C et en Java par les deux programmes de l'archive `RC4.zip`. Ces programmes, reproduits sur les figures 9 et 10, sont assez semblables. Ils utilisent tous les deux la même clef courte : les 5 octets de **`0x01`** à **`0x05`**. Néanmoins, le programme C manipule des octets codés par des **`unsigned char`** alors que le programme Java utilise des **`int`** (et non des **`Bytes`**), car Java ne permet pas d'additionner des **`Bytes`** et les conversions de **`byte`** vers **`int`** sont signées.

Question 1. Modifiez le `main()` de l'un de ces programmes afin de chiffrer le fichier `butokuden.jpg` de l'archive `HMAC.zip` en RC4, avec la clef « KYOTO ». Votre programme produira un fichier chiffré nommé `confidentiel.jpg`. Évidemment, la clef « KYOTO » doit être comprise comme une suite de 5 octets, codée en ASCII.

Question 2. Chiffrez en RC4 le fichier `confidentiel.jpg` avec la même clef.

Question 3. Comparez le fichier obtenu à la question précédente et le fichier `butokuden.jpg` original.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef LG_FLUX
#define LG_FLUX 10
#endif
// Ce programme ne produira par défaut que les 10 premiers octets de la clef longue.
typedef unsigned char uchar; // Les octets sont non-signés.

uchar clef[] = {0x01, 0x02, 0x03, 0x04, 0x05};

uchar state[256];
int i, j;
void initialisation(void);
uchar production(void);

int main(void) {
  initialisation();      // Phase d'initialisation de l'état du système RC4
  printf("Premiers octets de la clef longue : ");
  for (int k=0; k < LG_FLUX; k++) {
    printf("0x%02X ", production()); // Affichage d'un octet généré en hexadécimal
  }
  printf("\n");
  exit(EXIT_SUCCESS);
}

void echange (uchar *state, int i, int j) {
  uchar temp = state[i];
  state[i] = state[j];
  state[j] = temp;
}

void initialisation(void) {
  int lg = sizeof(clef);
  printf("Clef courte utilisée : ");
  for (int k = 0; k < lg ; k++ ) printf ("0x%02X ", clef[k]);
  printf("\nLongueur de la clef courte : %d\n",lg);
  for (i=0; i < 256; i++) state[i] = i;
  j = 0;
  for (i=0; i < 256; i++) {
    j = (j + state[i] + clef[i % lg]) % 256;
    echange(state,i,j);           // Echange des octets en i et j
  }
  i=0;
  j=0;
}

uchar production(void) {
  i = (i + 1) % 256;            // Incrémentation de i modulo 256
  j = (j + state[i]) % 256;     // Déplacement de j
  echange(state,i,j);           // Echange des octets en i et j
  return state[(state[i] + state[j]) % 256];
}

/*
  $ make
  gcc -o mon_RC4 -I/usr/local/include -I/usr/include mon_RC4.c -L/usr/local/lib
  -L/usr/lib -lm -lssl -lcrypto -g -Wall
  $ ./mon_RC4
  Clef courte utilisée : 0x01 0x02 0x03 0x04 0x05
  Longueur de la clef courte : 5
  Premiers octets de la clef longue : 0xB2 0x39 0x63 0x05 0xF0 0x3D 0xC0 0x27 0xCC 0xC3
*/
```

FIGURE 9 – Calcul d'une clef longue RC4 en C

```java
public class MonRC4
{
    private static int LG_FLUX = 10;
    // Ce programme ne produira que les 10 premiers octets de la clef longue.

    static int[] state = new int[256];  // Ces int seront tous positifs et <256
    static int i = 0, j = 0;            // car ils représentent l'ensemble des octets
    static int[] clef = {1, 2, 3, 4, 5};
    static int lg = clef.length;

    public static void main(String[] args) {
        initialisation();       // Phase d'initialisation de l'état du système RC4
        System.out.print("Premiers octets de la clef longue : ");
        for (int k = 0; k < LG_FLUX; k++) {
            System.out.printf("0x%02X ", production());
            // Affichage d'un octet généré en hexadécimal
        }
        System.out.print("\n");
    }

    private static void échange(int k, int l) {
        int temp = state[k];
        state[k] = state[l];
        state[l] = temp;
    }

    private static void initialisation() {
        System.out.print("Clef courte utilisée : ");
        for (int k = 0; k < lg ; k++ )
            System.out.printf("0x%02X ", clef[k]);
        System.out.println("\nLongueur de la clef courte : " + lg);
        for (i=0; i < 256; i++) state[i] = i;
        j = 0;
        for (int i=0; i < 256; i++) {
            j = (j + state[i] + clef[i % lg]) % 256;
            échange(i,j);               // Echange des octets en i et j
        }
        i = 0;
        j = 0;
    }

    private static int production() {
        i = (i + 1) % 256;          // Incrémentation de i modulo 256
        j = (j + state[i]) % 256;   // Déplacement de j
        échange(i,j);               // Echange des octets en i et j
        return state[(state[i] + state[j]) % 256];
    }
}
/*
  $ make
  javac *.java
  $ java MonRC4
  Clef courte utilisée : 0x01 0x02 0x03 0x04 0x05
  Longueur de la clef courte : 5
  Premiers octets de la clef longue : 0xB2 0x39 0x63 0x05 0xF0 0x3D 0xC0 0x27 0xCC 0xC3
*/
```

FIGURE 10 – Calcul d'une clef longue RC4 en Java

*Le principe de la production d'une clef longue à partir d'une clef courte apparaît également dans le système standard AES, que vous allez programmer intégralement.*

Le système de chiffrement symétrique AES utilise des clefs secrètes de longueur variable : 128, 192 ou 256 bits. La longueur d'une clé $k$ est caractérisée par le nombre $N_k$ de mots (de 32 bits) contenus dans la clef : $N_k$ vaut donc 4, 6 ou 8. Chacun de ces mots est vu comme un vecteur colonne de 4 octets. La clef $k$ est ainsi représentée par une matrice $K$ de 4 lignes et $N_k$ colonnes. Le placement des octets de la clef $k$ dans la matrice $K$ suit l'ordre illustré sur la figure 11.

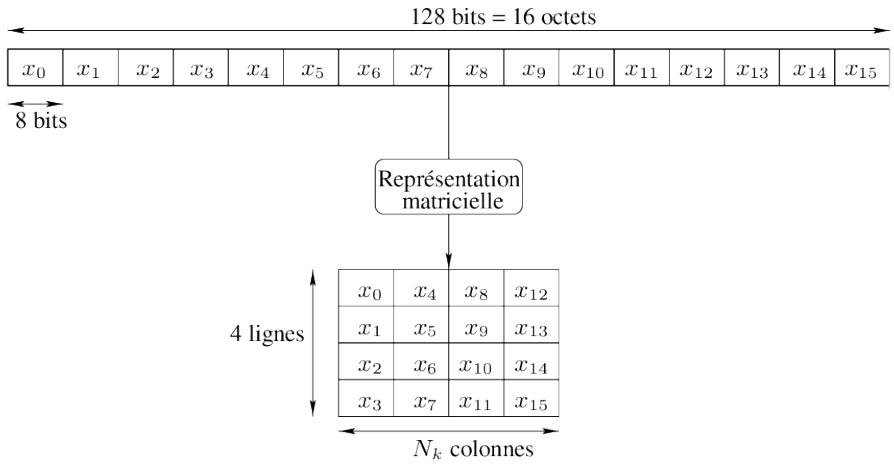


FIGURE 11 – Vue matricielle d'une clef AES (avec $N_k = 4$, c'est-à-dire une clef de 16 octets)

Lors de l'opération d'extension qui précède le chiffrement (ou le déchiffrement) la matrice $K$ est *étendue* en une matrice $W$ de 4 lignes et $4 \times (N_r + 1)$ colonnes, où le *nombre de rondes* $N_r$ suit les règles suivantes :

— $N_r = 10$ si $N_k = 4$ ;
— $N_r = 12$ si $N_k = 6$ ;
— $N_r = 14$ si $N_k = 8$.

Ainsi, plus la clef secrète $k$ est longue, plus le nombre de rondes associé est grand (c'est-à-dire que le chiffrement sera plus lent), et plus la clef étendue $W$ produite est large (Fig. 12). La matrice $W$ sera ensuite coupée en $N_r + 1$ morceaux formant chacun une matrice carrée de $4 \times 4$ octets, appelée *clef de ronde*.

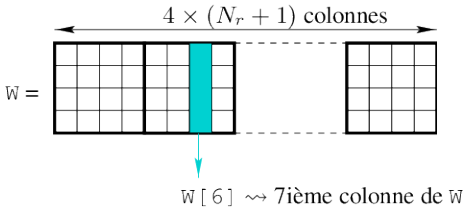


FIGURE 12 – Extension de la clef $K$ en une clef $W$ étendue (et $N_r + 1$ clefs de ronde)

**Extension de la clef**

Pour construire la matrice $W$, les $N_k$ colonnes de la clef $K$ sont tout d'abord recopiées *sans modification* dans les $N_k$ premières colonnes de $W$. Puis les colonnes suivantes de la matrice $W$ sont calculées les unes après les autres, selon l'algorithme KEYEXPANSION$(K, W)$ détaillé sur la figure 13. Dans cet algorithme, $K[i]$ désigne la $i$-ième colonne de la matrice $K$. C'est un vecteur formé de 4 octets. De même, $W[i]$ désigne la $i$-ième colonne de la matrice $W$. Ainsi, les deux premières lignes de l'algorithme KEYEXPANSION$(K, W)$ assurent simplement que la clef $K$ est recopiée intégralement dans les premières colonnes de la clef étendue $W$.

La variable auxilliaire $tmp$ est également un vecteur colonne de 4 octets. Chaque itération de la boucle **for** qui commence à la ligne 3 calcule la colonne $W[i]$ de la matrice $W$ et la stocke à la fin de la boucle, ligne

```
KeyExpansion(K, W)
 1  for i ← 0 to N_k − 1
 2  do W[i] ← K[i];                          // K est recopiée dans W colonne par colonne
 3  for i ← N_k to 4 × (N_r + 1) − 1
 4  do tmp ← W[i − 1];                       // La colonne précédente est recopiée dans tmp
 5     if i mod N_k = 0
 6       then tmp ← RotWord(tmp);           // Opération de confusion
 7            tmp ← SubWord(tmp);           // Opération de substitution
 8            tmp ← tmp ⊕ Rcon[i/N_k];
 9       else if (N_k > 6) and (i mod N_k = 4)
10              then tmp ← SubWord(tmp);
11     tmp ← W[i − N_k] ⊕ tmp;
12     W[i] ← tmp;
13  return
```

FIGURE 13 – Algorithme d'extension de la clef courte $K$ en une clef $W$ étendue

12 ; elle initialise ce calcul en recopiant la colonne précédente, $W[i-1]$, dans le vecteur auxilliaire $tmp$ (à la ligne 4). Puis le vecteur $tmp$ subit une série d'opérations (lignes 5 à 11) ; enfin, le vecteur colonne $tmp$ est recopié dans la colonne $W[i]$ (ligne 12). Notez ici que la condition de la ligne 5 est assez rarement satisfaite : elle requiert que le numéro $i$ de la colonne en cours de calcul soit un multiple de $N_k$. Hormis ce cas particulier, la seule transformation subie par $tmp$ est un XOR bit-à-bit avec le vecteur colonne $W[i-N_k]$ à la ligne 11. Ce XOR bit-à-bit s'applique donc systématiquement lors de la production de chaque nouvelle colonne de $W$. Ainsi, le plus souvent : $W[i] = W[i-N_k] \oplus W[i-1]$.

Observez aussi que l'algorithme diffère légèrement selon la longueur de la clef $k$. La condition $N_k > 6$ de la ligne 9 signifie en fait que $N_k = 8$ et donc que la clef $k$ comporte 256 bits. Dans ce cas seulement, la ligne 10 pourra effectivement être exécutée au cours des transformations subies par $tmp$ pour le calcul d'une nouvelle colonne, précisément lorsque $i$ est un multiple de 4.

**Opérations élémentaires sur la variable** $tmp$

Cet algorithme d'extension utilise les transformations élémentaires suivantes :

① La fonction **RotWord**, utilisée à la ligne 6, prend en entrée un mot, c'est-à-dire un vecteur colonne formé de 4 octets $[a_0, a_1, a_2, a_3]^\top$, et effectue une *rotation* de façon à renvoyer le vecteur colonne formé des octets $[a_1, a_2, a_3, a_0]^\top$. En particulier, l'octet $a_0$ situé en haut de la colonne se trouve replacé en bas. Les autres octets de la colonne montent d'un cran.

② La fonction **SubWord** appliquée à la ligne 7 effectue une substitution des octets du vecteur colonne $[a_0, a_1, a_2, a_3]^\top$ en entrée en renvoyant le vecteur $[s(a_0), s(a_1), s(a_2), s(a_3)]^\top$ où $s$ est une permutation sur l'ensemble des octets. Nous prendons pour acquis que la substitution $s$ utilisée par l'AES est décrite par le tableau de la figure 14. En particulier, $s(\texttt{0x00}) = \texttt{0x63}$, $s(\texttt{0x01}) = \texttt{0x7C}$ et $s(\texttt{0x62}) = \texttt{0xAA}$.

③ La dernière opération appliquée sur $tmp$ correspond à la ligne 8. Il s'agit d'effectuer à nouveau un XOR bit-à-bit avec un autre vecteur noté $\texttt{Rcon}[i/N_k]$. Remarquez que cette opération n'est appliquée que si $i \mod N_k = 0$, c'est-à-dire que $i/N_k$ est un entier. La valeur minimale de cet entier est $N_k/N_k = 1$. Sa valeur maximale vaut $(4 \times (N_r + 1) - 1)/N_k$, qui est au plus égal à 10 quel que soit la longueur de la clef $k$. Nous prendrons pour acquis que les vecteurs colonnes $\texttt{Rcon}[1]$,..., $\texttt{Rcon}[10]$ sont donnés par la formule suivante :

$$\texttt{Rcon}[j] = \begin{bmatrix} x_j \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

et que les valeurs de l'octet $x_j$, pour $j$ allant de 1 à 10, sont les suivantes :

```
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
```

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

FIGURE 14 – Table de substitution (appelée SBox) utilisée pour **SubWord**

**Exercice B.2 Un exemple sur le papier** Nous choisissons dans cet exercice d'utiliser la clef nulle $K =$ **`0x00 0x00 0x00 ... 0x00`** formée de 128 bits, c'est-à-dire 16 octets. Les 20 premiers octets de la clef étendue $W$ sont indiqués sur le tableau ci-dessous.

Question 1. Expliquez pourquoi la cinquième colonne vaut effectivement (**`0x62`**, **`0x63`**, **`0x63`**, **`0x63`**).

Question 2. Calculez les 16 octets suivants de $W$ et notez-les ci-dessous.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x62$ | | | | | |
| $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x63$ | | | | | |
| $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x63$ | | | | | |
| $0x00$ | $0x00$ | $0x00$ | $0x00$ | $0x63$ | | | | | |

**Exercice B.3 Extension de la clef courte dans l'AES** Dans cet exercice, il vous faut compléter le programme `diversification.c`, décrit sur la figure 15, ou le programme `Diversification.java` équivalent, tous deux disponibles dans l'archive `AES.zip`. Ces programmes définissent les tableaux d'octets suivants :

- **`Sbox[256]`** qui contient la table de substitution $s$ de l'opération **SubWord** (fig. 14) ;
- **`K[32]`** qui contient la clef courte $K$ sous la forme d'une suite de 16, 24 ou 32 octets ;
- **`W[240]`** qui devra contenir la clef étendue $W$.

De plus, les variables entières **`Nk`** et **`Nr`** représentent la longueur $N_k$ de la clef courte $K$, en mots, et le nombre de rondes $N_r$ associées. Ce squelette de programme est disponible dans l'archive `AES.zip` sur le site de l'UE.

Question 1. Écrire le code de la fonction **`calcule_la_clef_courte()`** afin d'analyser la clef fournie en paramètre et en hexadécimal afin de construire la clef courte représentée par le tableau **`K`**.

Question 2. Écrire le code de la fonction **`calcule_la_clef_etendue()`** afin de remplir le tableau **`W`** à partir de la clef courte **`K`** selon l'algorithme KEYEXPANSION.

Question 3. Tester le programme sur la clef courte formée de 16 octets nuls, en comparant les premiers octets au résultat obtenu à l'exercice précédent.

Question 4. Tester le programme sur la clef courte formée des 16 octets **`0x2B`**, **`0x7E`**, **`0x15`**, **`0x16`**, **`0x28`**, **`0xAE`**, **`0xD2`**, **`0xA6`**, **`0xAB`**, **`0xF7`**, **`0x15`**, **`0x88`**, **`0x09`**, **`0xCF`**, **`0x4F`**, **`0x3C`**, en comparant le résultat obtenu à celui indiqué en commentaire dans le fichier `diversification.c` fourni.

Question 5. Vérifier également que la clef courte de 32 octets

$$k = \texttt{0xff 0xff 0xff 0xff ... 0xff}$$

correspond à la clef longue indiquée sur la figure 16.

```c
typedef unsigned char uchar;

uchar SBox[256] = { 0x63, 0x7C, 0x77, 0x7B, 0xF2, ... };        // Table de substitution
uchar Rcon[10] = { 0x01, 0x02, 0x04, 0x08, ...};                  // Constantes de ronde
uchar K[32];                       // Une clef courte a une longueur maximale de 32 octets
uchar W[240];          // La clef longue a une longueur maximale de (14+1)*16=240 octets
int longueur_de_la_clef, longueur_de_la_clef_etendue, Nk, Nr;

void affiche_la_clef(uchar *clef, int longueur) {
  for (int i=0; i<longueur; i++) { printf ("%02X ", clef[i]); }
  printf("\n");
}

int main(int argc, char* argv[]) {
  if(argc != 2){
    printf("Usage: %s <clef courte AES, écrite en hexadécimal>", argv[0]);
    exit(EXIT_FAILURE);
  }

  longueur_de_la_clef = strlen(argv[1])>>1; // Il faut 2 caractères pour faire un octet!

  if((longueur_de_la_clef!=16)&&(longueur_de_la_clef!=24)&&(longueur_de_la_clef!=32)) {
    printf("Usage: %s <clef courte AES, écrite en hexadécimal>\n", argv[0]);
    printf("\t Une clef AES est formée de 32, 48, ou 64 caractères hexadécimaux,\n");
    printf("\t c'est-à-dire 128, 192, ou 256 bits.\n");
    exit(EXIT_FAILURE);
  }

  calcule_la_clef_courte(argv[1]);    // Fonction décodant la clef courte K
  calcule_la_clef_etendue();          // Fonction calculant la clef longue W

  affiche_la_clef(W, longueur_de_la_clef_etendue);
  exit(EXIT_SUCCESS);
}
```

FIGURE 15 – Programme `diversification.c` à compléter

```
W = ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
    e8 e9 e9 e9 17 16 16 16 e8 e9 e9 e9 17 16 16 16
    0f b8 b8 b8 f0 47 47 47 0f b8 b8 b8 f0 47 47 47
    4a 49 49 65 5d 5f 5f 73 b5 b6 b6 9a a2 a0 a0 8c
    35 58 58 dc c5 1f 1f 9b ca a7 a7 23 3a e0 e0 64
    af a8 0a e5 f2 f7 55 96 47 41 e3 0c e5 e1 43 80
    ec a0 42 11 29 bf 5d 8a e3 18 fa a9 d9 f8 1a cd
    e6 0a b7 d0 14 fd e2 46 53 bc 01 4a b6 5d 42 ca
    a2 ec 6e 65 8b 53 33 ef 68 4b c9 46 b1 b3 d3 8b
    9b 6c 8a 18 8f 91 68 5e dc 2d 69 14 6a 70 2b de
    a0 bd 9f 78 2b ee ac 97 43 a5 65 d1 f2 16 b6 5a
    fc 22 34 91 73 b3 5c cf af 9e 35 db c5 ee 1e 05
    06 95 ed 13 2d 7b 41 84 6e de 24 55 9c c8 92 0f
    54 6d 42 4f 27 de 1e 80 88 40 2b 5b 4d ae 35 5e
```

FIGURE 16 – Test du programme `diversification.c` complété

```java
public class Diversification {

  public static byte[] SBox = { (byte) 0x63, (byte) 0x7C, (byte) 0x77, ... };
  public static byte[] Rcon = { (byte) 0x01, (byte) 0x02, (byte) 0x04, ... } ;
  public static byte K[] = new byte[32];
  public static byte W[] = new byte[240];
  public static int longueur_de_la_clef;
  public static int longueur_de_la_clef_etendue;
  public static int Nr;
  public static int Nk;

  public static void affiche_la_clef(byte clef[], int longueur) {
    for (int i=0; i<longueur; i++) System.out.printf ("%02X ", clef[i]);
    System.out.println();
  }

  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("Usage: java Diversification <clef en hexadécimal>");
      System.exit(1);
    }

    longueur_de_la_clef = args[0].length()/2;

    if ( (longueur_de_la_clef!=16) && (longueur_de_la_clef!=24)
         && (longueur_de_la_clef != 32) ) {
      System.out.println("Usage: java Diversification <clef en hexadécimal>");
      System.out.println("\t Une clef AES est formée de 32, 48, ou 64 chiffres,");
      System.out.println("\t c'est-à-dire 128, 192, ou 256 bits.");
      System.exit(1);
    }
    calcule_la_clef_courte(args[0]);     // Fonction décodant la clef courte K
    calcule_la_clef_etendue();           // Fonction calculant la clef longue W
    affiche_la_clef(W, longueur_de_la_clef_etendue);
  }

  public static void calcule_la_clef_courte(String clef) {} // À modifier !
  public static void calcule_la_clef_etendue() {} // À modifier !
}


/* 2nd test à effectuer:
   $ java Diversification 2b7e151628aed2a6abf7158809cf4f3c
   La clef est : 2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C
   Les clefs de rondes sont :
   RoundKeys[00] =  2B  7E  15  16  28  AE  D2  A6  AB  F7  15  88  09  CF  4F  3C
   RoundKeys[01] =  A0  FA  FE  17  88  54  2C  B1  23  A3  39  39  2A  6C  76  05
   RoundKeys[02] =  F2  C2  95  F2  7A  96  B9  43  59  35  80  7A  73  59  F6  7F
   RoundKeys[03] =  3D  80  47  7D  47  16  FE  3E  1E  23  7E  44  6D  7A  88  3B
   RoundKeys[04] =  EF  44  A5  41  A8  52  5B  7F  B6  71  25  3B  DB  0B  AD  00
   RoundKeys[05] =  D4  D1  C6  F8  7C  83  9D  87  CA  F2  B8  BC  11  F9  15  BC
   RoundKeys[06] =  6D  88  A3  7A  11  0B  3E  FD  DB  F9  86  41  CA  00  93  FD
   RoundKeys[07] =  4E  54  F7  0E  5F  5F  C9  F3  84  A6  4F  B2  4E  A6  DC  4F
   RoundKeys[08] =  EA  D2  73  21  B5  8D  BA  D2  31  2B  F5  60  7F  8D  29  2F
   RoundKeys[09] =  AC  77  66  F3  19  FA  DC  21  28  D1  29  41  57  5C  00  6E
   RoundKeys[10] =  D0  14  F9  A8  C9  EE  25  89  E1  3F  0C  C8  B6  63  0C  A6
   $
*/
```

FIGURE 17 – Programme `Diversification.java` à compléter