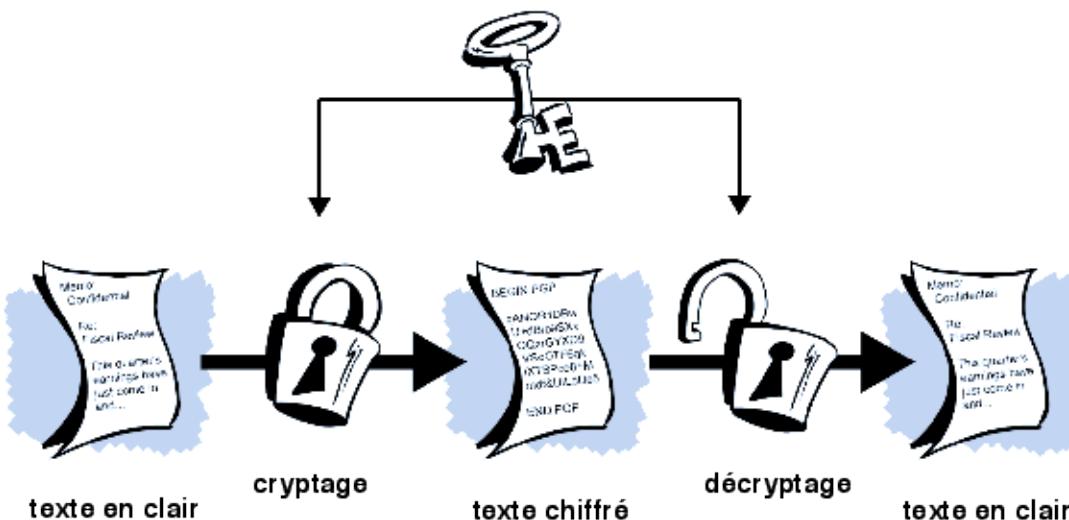


Cryptographie symétrique par flot

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

Plan des deux heures de cours à venir



① Les schémas cryptographiques symétriques par flots

- ~~> Le système RC4
- ~~> Les registres à décalage linéaire (LFSR)

② Les schémas cryptographiques symétriques par blocs

- ~~> Le bourrage de fichiers
- ~~> Les modes opératoires

③ Présentation technique de l'AES

Chiffrement à masque jetable (Gilbert Vernam, 1890-1960)

Système à masque jetable ou « one-time-pad »

L'opération de chiffrement d'un message $M = m_1 \dots m_n$ formé de n bits est :

$$C = E_K(M) = M \oplus K = m_1 \oplus k_1 \dots m_n \oplus k_n$$

La clé secrète K est donc une suite $K = k_1 \dots k_n$ formée également de n bits.

$$\begin{array}{r} M = 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\ \oplus \quad K = 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \\ \hline C = 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array}$$

Le déchiffrement consiste en la même opération car $(m \oplus k) \oplus k = m$, quels que soient les bits m et k .

Chiffrement à masque jetable (Gilbert Vernam, 1890-1960)

Exemples. Selon certaines sources, le Kremlin et la Maison Blanche utilisaient ce système pour communiquer. Un chiffrement à la main par la méthode du masque jetable fut aussi utilisé par Che Guevara pour communiquer avec Fidel Castro.

Avec ce système, la clef ne doit être utilisée en principe qu'**une seule fois** afin de résister aux *attaques à clairs connus*.

Pourquoi ?

Justification

L'attaque à clair connu suppose que l'adversaire connaisse un message $M = m_1 \dots m_n$ et son chiffré $C = c_1 \dots c_n$.

Comme $c_i = m_i \oplus k_i$, on a $m_i \oplus c_i = m_i \oplus (m_i \oplus k_i) = k_i$.

Ainsi on peut retrouver la clef K facilement.

Si on utilise une seconde fois la même clef, l'adversaire pourra déchiffrer le texte chiffré.

Chiffrements par flots

Les schémas de chiffrement par flot traitent l'information bit par bit (ou octet par octet) et sont extrêmement rapides. Ils peuvent être implémentés avec une mémoire réduite. De plus, la propagation des erreurs de transmission du chiffré au clair est limitée.

Ils sont parfaitement adaptés à des moyens de calcul, de mémoire et de transmission contraints (**cryptographie en temps réel**) : cryptographie militaire, ou cryptographie entre le téléphone portable GSM et son réseau (système A5/1).

Leur principe est d'effectuer un **chiffrement de Vernam** en utilisant une **clé longue** générée à partir d'une clef, appelée **clef courte**, d'une longueur suffisante pour résister aux attaques exhaustives.

Ceci nécessite des clefs courtes d'au moins 128 bits.

La clef longue est aussi appelée **clé pseudo-aléatoire**.

✓ *Les schémas cryptographiques par flots*

👉 *Exemple du RC4*

Premier exemple de générateur aléatoire très utilisé : RC4

RC4 est un algorithme de chiffrement à flot conçu en 1987 par Ronald Rivest, l'un des inventeurs du RSA. Il est utilisé pour le WEP, le WPA (par défaut), les protocoles de cryptage proposés dans BitTorrent, les documents PDF, etc. Il est aussi disponible en option dans SSL, SSH, etc.

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 courte** C secrète, 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.

Initialisation du tableau à partir de la clef courte

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
-------	------	------	------	-----	------

Les éléments du tableau S sont ensuite mélangés grâce à la clef courte C , de taille L quelconque (idéalement 256 octets), vue comme un tableau d'octets : $C[0] \dots C[L-1]$. Ce mélange initial opère les 256 échanges suivants :

$j := 0$

pour i de 0 à 255

$j := (j + S[i] + C[i \bmod L]) \bmod 256$

échanger($S[i], S[j]$)

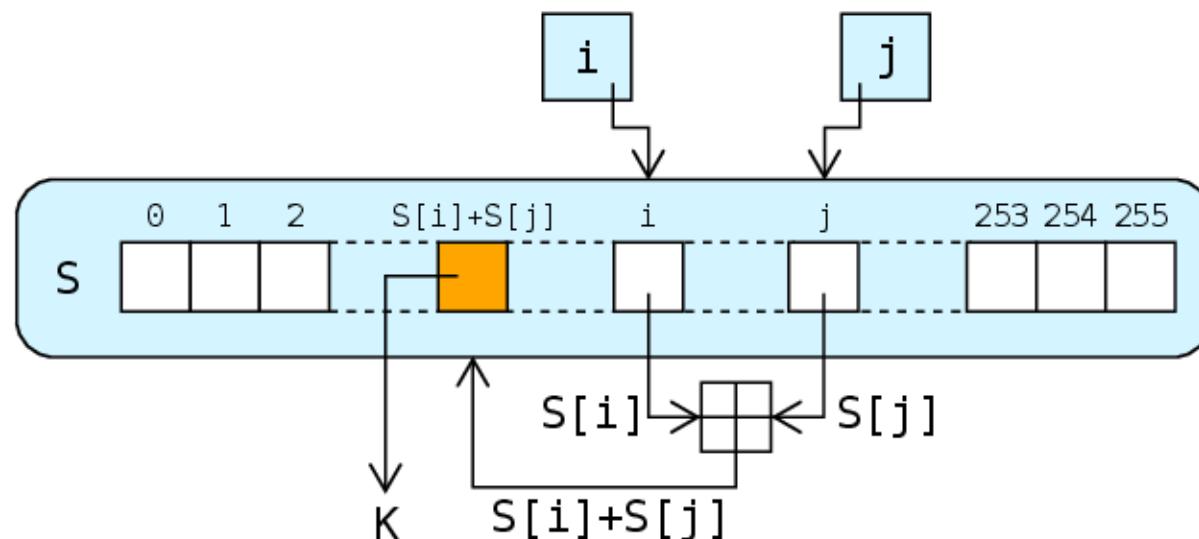
fin pour

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 ($L \leq 255$), alors « on la répète » (c'est le rôle du $\bmod L$).

Production d'un octet de la clef longue

Pour produire un octet de la clef longue, il suffit de choisir un octet particulier dans le tableau S ; ce choix repose sur les valeurs 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 le tableau contenant l'octet produit.



$$K := S[(S[i] + S[j]) \bmod 256]$$

Modification de l'état interne lors de la production d'un octet

Avant la production d'un nouvel octet de la clef longue, l'état interne est modifié selon les trois instructions suivantes :

```
i := (i + 1) mod 256      // Déplacement de i  
j := (j + S[i]) mod 256 // Déplacement de j  
échanger(S[i], S[j])    // Echange dans le tableau S
```

Ainsi

- les pointeurs *i* et *j* se déplacent;
- le contenu du tableau *S* est modifié.

La suite d'octets générée en guise de clef longue ne dépend que de la clef courte C ; celle-ci n'intervient que dans la phase d'initialisation de l'état interne.

Code à utiliser en Travaux Pratiques : calcul 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();      // Initialisation de l'état du système RC4  
    printf("Premiers_octets_de_la_clef_longue:_");  
    for (int k=0; k < 10; k++) {  
        printf("0x%02X_", production()); // Affichage de l'octet généré  
    }  
    printf("\n");  
}
```

- ✓ *Les schémas cryptographiques par flots*
- ✓ *Exemple du RC4*
- 👉 *Les registres à décalage linéaire (LFSR)*

Les registres à décalage à rétroaction linéaire

Dans un **registre à décalage à rétroaction linéaire** (en anglais : «linear feedback shift register» ou LFSR) la suite pseudo-aléatoire $S = s_0 \dots s_n \dots$ formant les bits de la **clef longue** est obtenue par une relation de récurrence linéaire du type suivant :

$$s_j = \sum_{k=1}^L c_k \times s_{j-k} \mod 2$$

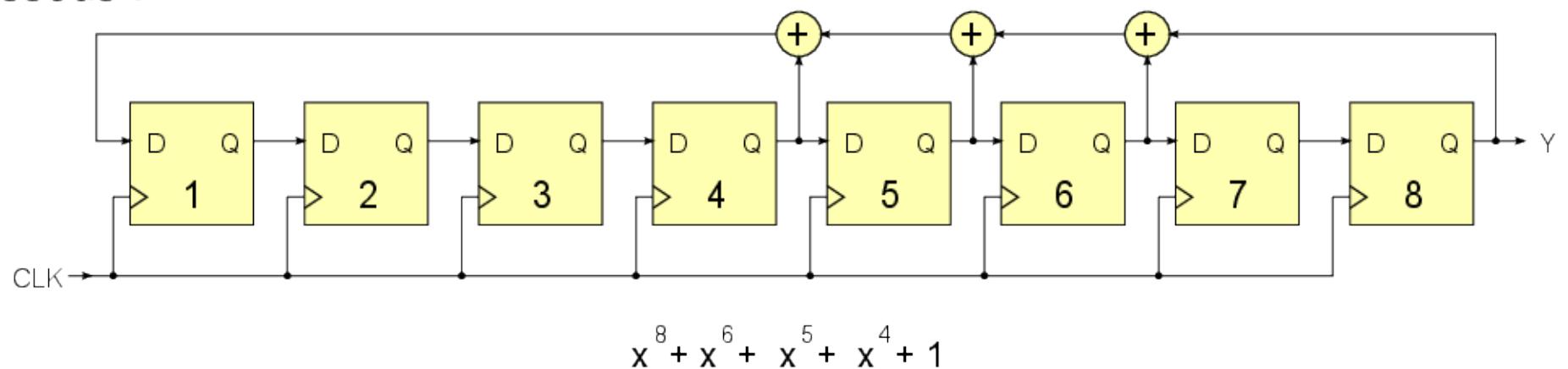
Exemple : Avec $L=8$, nous pourrions adopter $s_j = s_{j-4} \oplus s_{j-5} \oplus s_{j-6} \oplus s_{j-8}$.

Les L premiers bits s_0, \dots, s_{L-1} constituent la **clef courte** et déterminent entièrement la **clef longue** S produite à l'aide des constantes binaires c_1, \dots, c_L fixées.

Implémentation simple des registres à décalage à rétroaction linéaire

Le calcul du bit s_{i+L} s'effectue facilement à l'aide d'un **circuit synchrone** muni de L bascules D qui conservent les valeurs des L derniers bits produits :

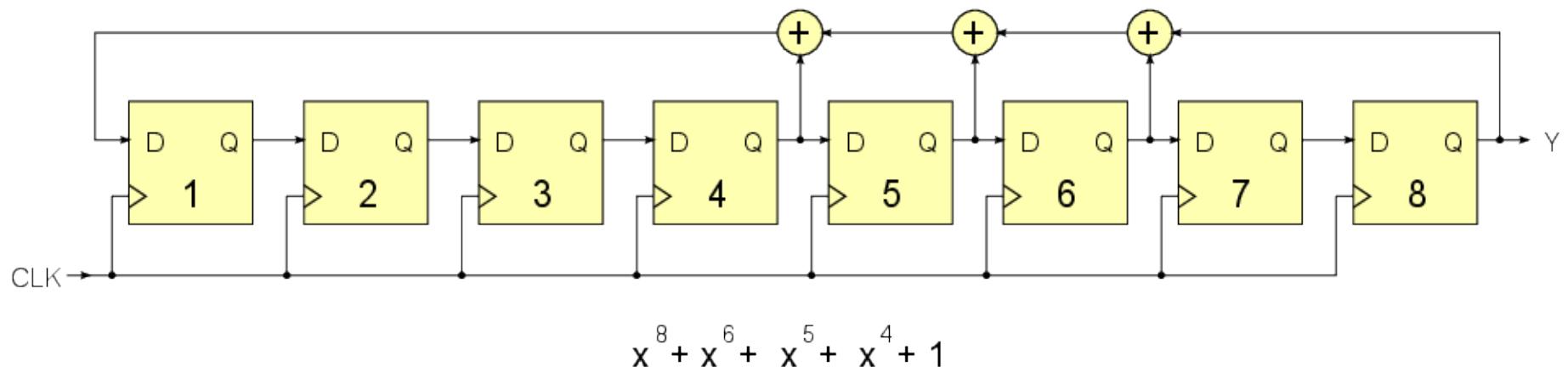
Exemple : Le LFSR défini par $s_j = s_{j-4} \oplus s_{j-5} \oplus s_{j-6} \oplus s_{j-8}$ correspond au circuit ci-dessous :



Polynôme de rétroaction

Les bits constants c_1, \dots, c_k sont habituellement représentés par un polynôme de degré L, appelé *polynôme de rétroaction du registre* :

$$P(X) = 1 + c_1.X + c_2.X^2 + \dots + c_L.X^L$$



LFSR combinés

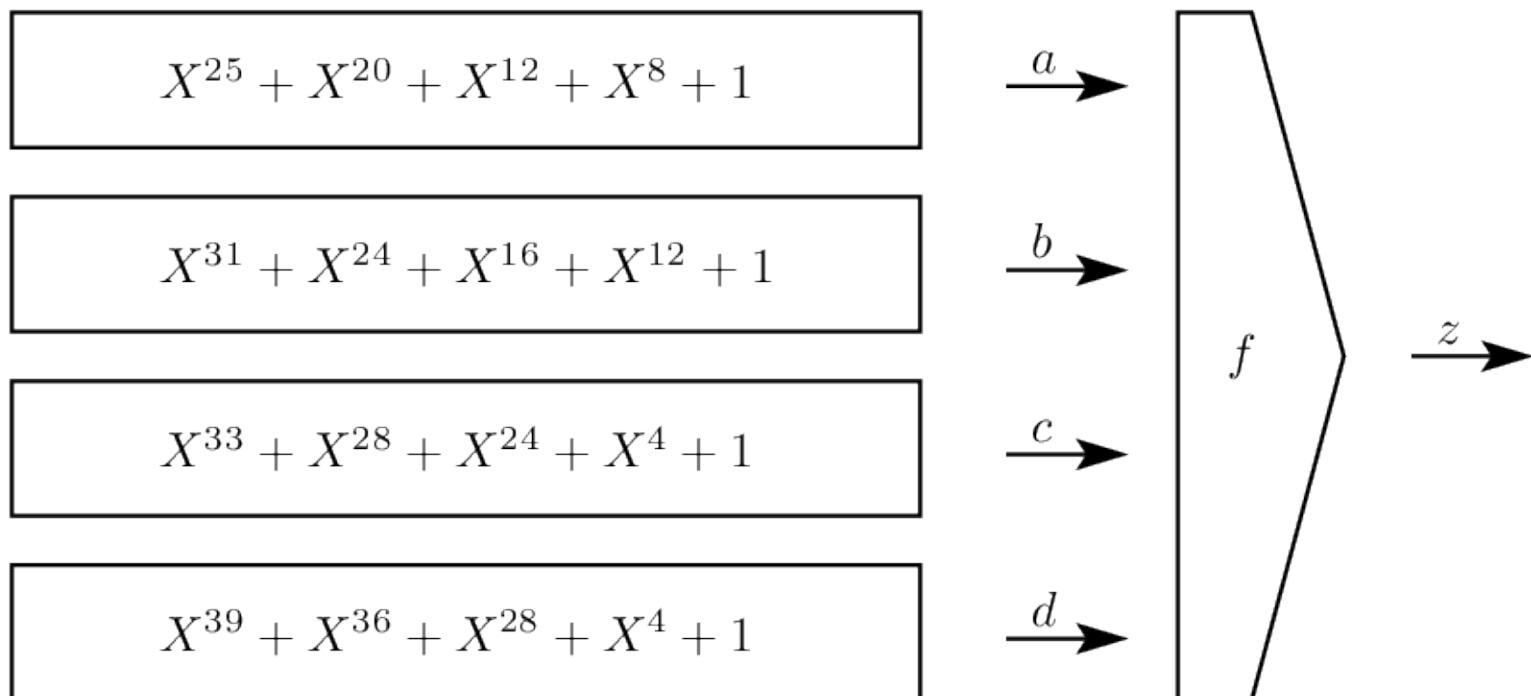
Les LFSR ne sont pas cryptographiquement sûrs à cause de leur structure linéaire.

En 1969, J. Massey a montré comment retrouver le polynôme de rétroaction d'un LFSR à partir uniquement des $2 \times L$ premiers bits de la suite produite.

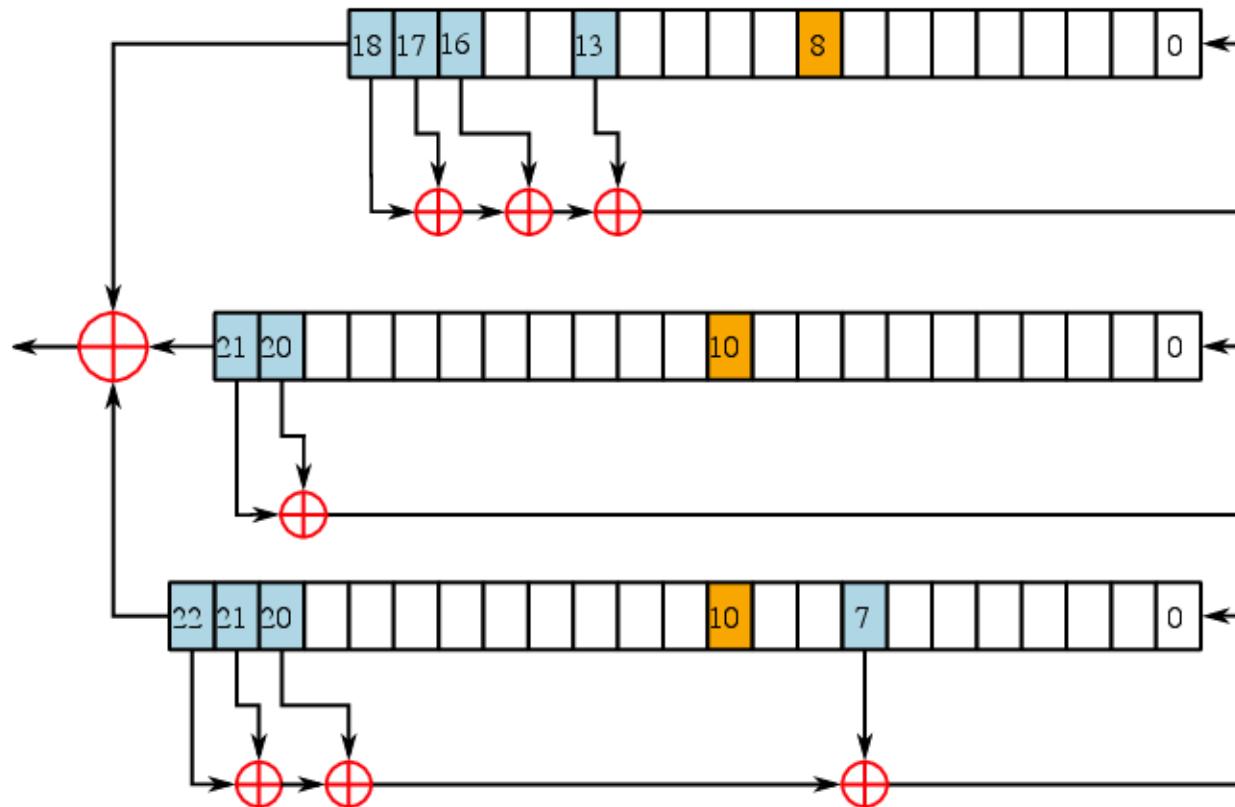
Si on devine le début du message clair et si on dispose du message chiffré, alors on obtient aisément le début de la clef, et par conséquent potentiellement toute la clef.

LFSR combinés

Un renforcement de la méthode LFSR consiste à associer plusieurs LFSR à l'aide d'une fonction booléenne **non linéaire** f .



A5/1, le chiffrement du GSM en Europe



Sur chaque période, le bit de sortie, à gauche, est le XOR des bits de sortie des 3 LFSR.

Un registre effectue un décalage au front montant si son bit orange coïncide avec la **majorité** des trois bits orange (le 8ième du premier et le 10ième des deux autres) : il y a donc sur chaque période au moins 2 des 3 registres qui effectuent un décalage.

Comme indiqué sur le schéma, le bit inséré à droite correspond alors au XOR des bits en bleu.

Le standard AES

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

AES : Advanced Encryption Standard

Le standard de chiffrement avancé (**Advanced Encryption Standard ou AES**), aussi connu sous le nom de RIJNDAEL, est un algorithme de chiffrement **symétrique**, choisi en octobre 2000 par le NIST pour être le *nouveau standard de chiffrement pour les organisations du gouvernement des États-Unis*.

Il est issu d'un appel à candidatures international lancé en janvier 1997 et ayant reçu 15 propositions. Parmi ces 15 algorithmes, 5 furent choisis pour une évaluation plus poussée en avril 1999 : MARS, RC6, Rijndael, Serpent, et Twofish. Au bout de cette évaluation, ce fut finalement le candidat Rijndael, du nom de ses deux concepteurs Joan Daemen et Vincent Rijmen (tous les deux de nationalité belge) qui a été choisi.

AES est un sous-ensemble de Rijndael : il ne travaille qu'avec des blocs de 128 bits alors que Rijndael offre des tailles de blocs qui sont des multiples de 32 bits, compris entre 128 et 256 bits.

Fonctionnement de l'AES

Il s'agit d'un chiffrement consistant à opérer des opérations de **substitution** et de **permutation** *de manière répétée* : à chaque itération (appelée *ronde*), le texte chiffré produit par le tour précédent subit une substitution (non-linéaire) qui assure la **confusion** puis une permutation (linéaire) qui assure la **diffusion** ; enfin la *clef du ronde* est ajoutée bit-à-bit (XOR).

Les clefs de ronde sont fabriquées à partir de la clef secrète, appelée clef courte.

Mathématiquement, chaque octet est vu comme un élément du corps fini \mathbb{F}_{256} à 256 éléments.

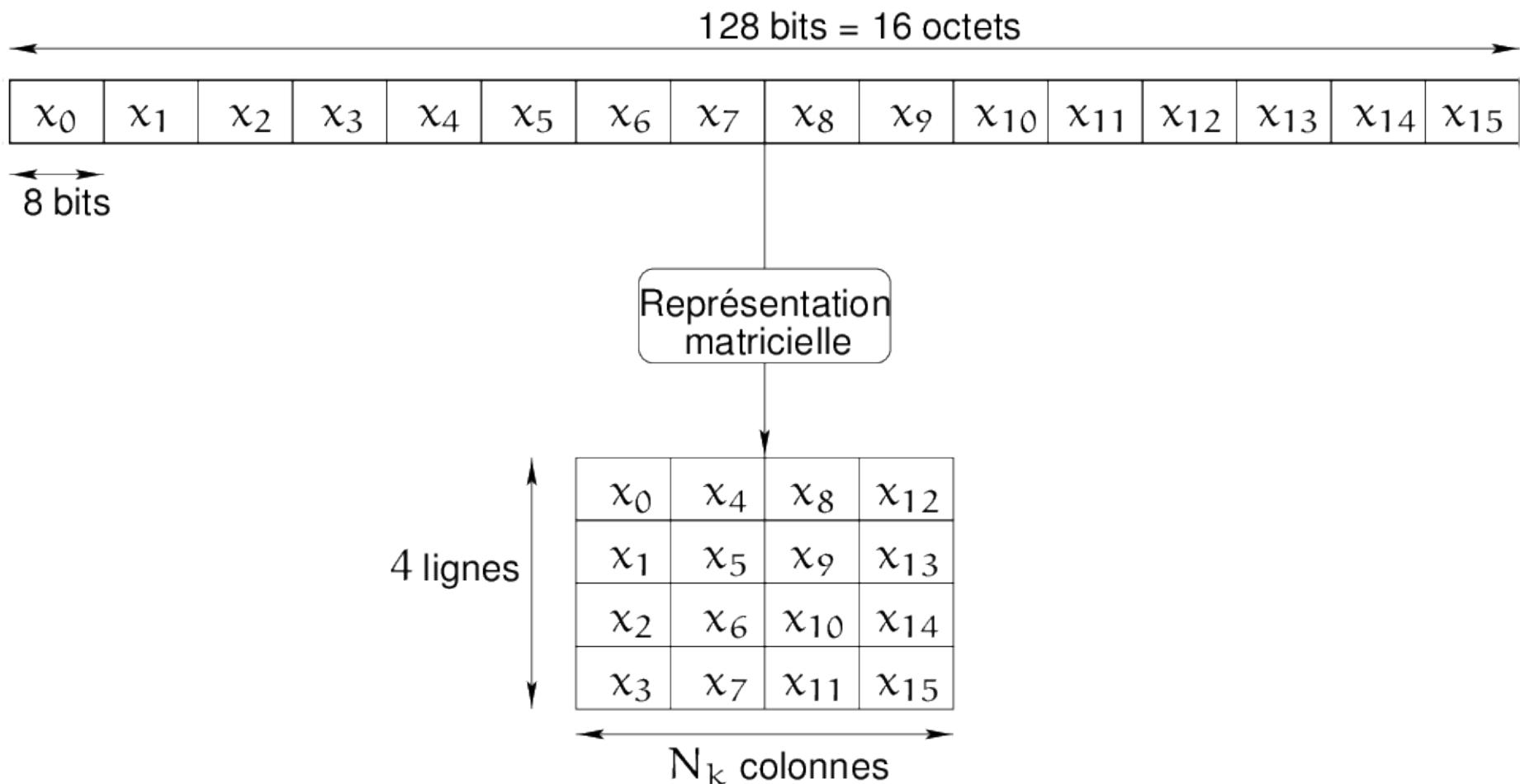
- L'addition de 2 octets correspond à un XOR bit-à-bit.
- La multiplication correspond à un produit de polynômes binaires dont on conserve le reste après une division euclidienne par un polynôme fixé, de degré 8.

Dans la littérature, le corps \mathbb{F}_{256} est aussi noté $GF(2^8)$ en hommage au mathématicien Evariste Galois (GF=Galois Field).

Clefs secrètes de l'AES

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.

La clef k peut donc être représentée par une matrice K de 4 lignes et N_k colonnes.





Clefs de ronde

Nombre de rondes et clefs de ronde

Comme le DES, son prédecesseur, l'AES exécute pour chiffrer un bloc de données une répétition de *rondes* : le nombre N_r de rondes qui doivent être effectuées lors du chiffrement (ou du déchiffrement) dépend de la valeur de N_k :

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

Ainsi, plus la clef est longue, plus il y a d'itérations (rondes).

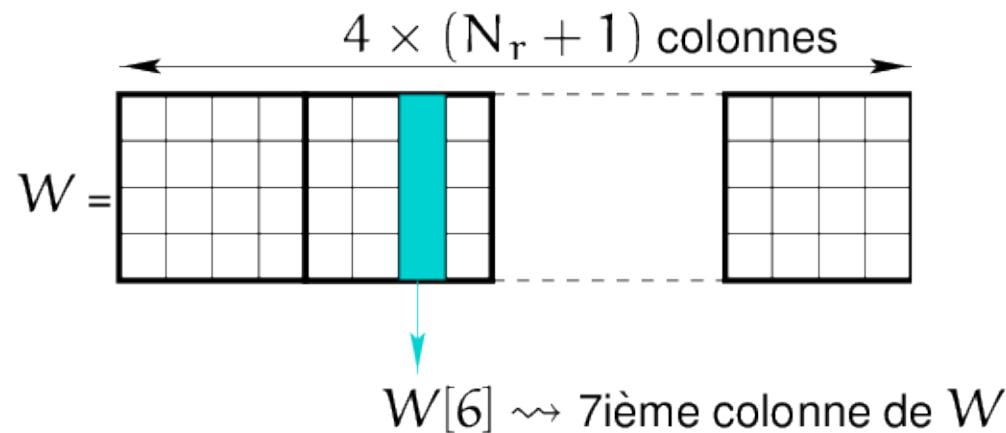
D'autre part, chaque ronde utilise une clef de ronde, notée **RoundKeys [i]**, vue comme une matrice 4×4 d'octets.

Ces **clefs de ronde** sont simplement des morceaux extraits d'une **clef longue** construite à partir de la clef secrète.

Extension de la clef secrète en clefs de ronde

Pour construire ces N_r clefs de rondes, la matrice K de 4 lignes et N_k colonnes est *étendue* en une matrice W de 4 lignes et $4 \times (N_r + 1)$ colonnes.

Pour construire cette 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.



Enfin, la matrice W est découpée en $N_r + 1$ morceaux qui forment chacun une matrice carrée de 4×4 octets : ce sont les *clefs de ronde*.

Extension de la clef courte : de K à W

KEYEXPANSION(K,W)

```

1 for  $i \leftarrow 0$  to  $N_k - 1$ 
2 do  $W[i] \leftarrow K[i];$  //  $K$  est recopiée dans  $W$  colonne par colonne
3 for  $i \leftarrow N_k$  to  $4 \times (N_r + 1) - 1$ 
4 do  $tmp \leftarrow W[i - 1];$  // La colonne précédente est recopiée dans  $tmp$ 
5   if  $i \bmod N_k = 0$ 
6     then  $tmp \leftarrow \text{RotWord}(tmp);$  // Opération de confusion
7      $tmp \leftarrow \text{SubWord}(tmp);$  // Opération de substitution
8      $tmp \leftarrow tmp \oplus Rcon[i/N_k];$ 
9   else if ( $N_k > 6$ ) and ( $i \bmod N_k = 4$ )
10     then  $tmp \leftarrow \text{SubWord}(tmp);$ 
11    $tmp \leftarrow W[i - N_k] \oplus tmp;$ 
12    $W[i] \leftarrow tmp;$  // Ajout de 4 octets
13 return

```

Explications : recopie de K dans W

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 $\text{KEYEXPANSION}(K, W)$ assurent simplement que la clef K est recopiée intégralement dans les premières colonnes de la clef étendue W .

Explications : itération colonne par colonne

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 et ajoute à `W` la nouvelle colonne `W[i]`.

Le calcul de la colonne `W[i]` commence par recopier 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).

Explications : cas simple, le plus fréquent

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 : $W[i] = W[i - N_k] \oplus W[i - 1]$.

Explications : cas des clefs courtes de 256 bits

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.

Opérations élémentaires : RotWord

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 $[a_1, a_2, a_3, a_0]^\top$.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \rightsquigarrow \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_0 \end{bmatrix}$$

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.

Opérations élémentaires : SubWord

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. La table de substitution (ou SBox) utilisée par l'AES est la suivante :

	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

En particulier, $s(0x00) = 0x63$, $s(0x01) = 0x7C$ et $s(0x62) = 0xAA$.

Opérations élémentaires : ajout de la constante de ronde

La ligne 8 effectue un XOR bit-à-bit entre `tmp` et un autre vecteur noté `Rcon[i/Nk]`.

Lorsque cette opération est appliquée, $i \bmod N_k = 0$ et donc 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 ≤ 10 .

Les vecteurs colonnes `Rcon[1],...,Rcon[10]` sont de la forme

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

où les constantes x_j , pour j allant de 1 à 10, sont les suivantes :

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

Travaux pratiques : un programme à compléter

```
// Table de substitution
uchar SBox[256] = { 0x63, 0x7C, 0x77, 0x7B, 0xF2, ... };
// Constantes de ronde
uchar Rcon[10] = { 0x01, 0x02, 0x04, 0x08, ... };
// Une clef courte a une longueur maximale de 32 octets
uchar K[32];
// La clef longue a une longueur maximale de (14+1)*16=240 octets
uchar W[240];
int longueur_de_la_clef, longueur_de_la_clef_etendue, Nk, Nr;
int main(int argc, char* argv[]) {
    // Il faut 2 caractères hexadécimaux pour faire un octet...
    longueur_de_la_clef = strlen(argv[1])>>1;
    calcule_la_clef_courte(argv[1]); // Décodage la clef courte K
    calcule_la_clef_etendue();      // Calcul de la clef longue W
    affiche_la_clef(W, longueur_de_la_clef_etendue);
}
```

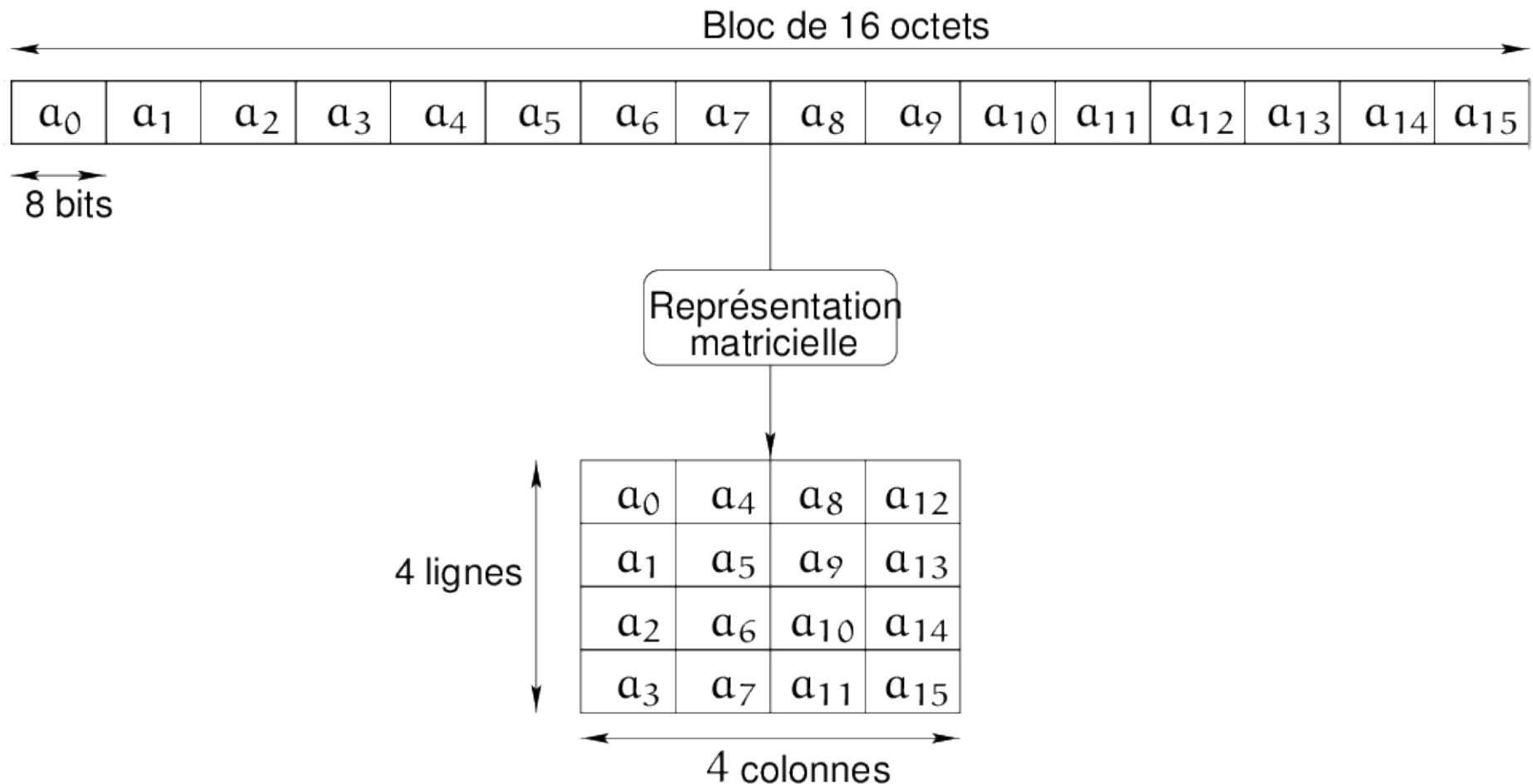
✓ *Clefs de ronde*

👉 *Chiffrement et déchiffrement*

Chiffrement de l'AES

L'AES est un système cryptographique symétrique par blocs.

Chaque bloc comporte 128 bits, c'est-à-dire 16 octets, vus comme une matrice 4×4 appelée *State*



Approche algébrique des octets

L'AES applique sur les octets deux opérations : l'addition, qui correspond à un XOR bit-à-bit, notée \oplus ou \oplus , et une multiplication notée \times . Toutes les deux sont *commutatives*.

L'octet nul $0x00$ est l'élément neutre pour l'addition : $0x00 \oplus a = a \oplus 0x00 = a$.

De plus, chaque octet est son propre opposé : $a \oplus a = 0x00$.

$0x01$ apparaîtra comme l'élément neutre de la multiplication : $0x01 \times a = a \times 0x01 = a$.

De plus, l'octet nul $0x00$ est **absorbant** : $0x00 \times a = a \times 0x00 = 0x00$.

Mathématiquement, les octets munis de ces deux opérations forment un *corps* noté \mathbb{F}_{256} .

En particulier, tout octet a possède un inverse b , c'est-à-dire que $a \times b = b \times a = 0x01$.

Les tables de multiplications seront données dans le squelette de programme à compléter en Travaux Pratiques.

Chiffrement par itérations de rondes

Le chiffrement consiste à opérer une série de transformations sur le bloc. Le chiffré du bloc initial est le bloc obtenu à la fin de ces transformations.

L'une d'entre elle fait intervenir les $N_r + 1$ *clefs de rondes*, de 0 à N_r , extraites de la clef étendue.

AES-ENCRYPT(State, K)

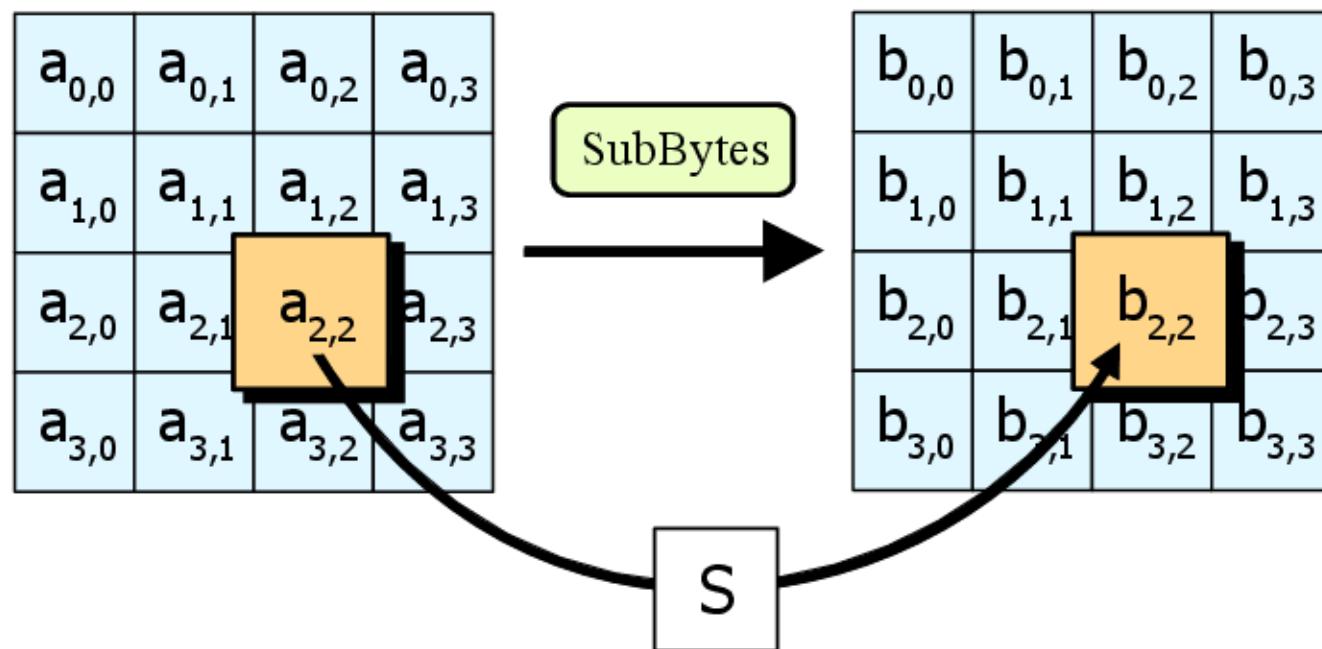
- 1 **KeyExpansion**(K, RoundKeys);
- 2 **AddRoundKey**(State, RoundKeys[0]);
- 3 **for** $i \leftarrow 1$ **to** $N_r - 1$
- 4 **do** // Boucle de $N_r - 1$ rondes constituées chacune de 4 opérations
- 5 **SubBytes**(State);
- 6 **ShiftRows**(State);
- 7 **MixColumns**(State);
- 8 **AddRoundKey**(State,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

Les quatre étapes 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*. Puis $N_r - 1$ **rondes** sont effectuées, chacune constituée de quatre transformations élémentaires sur le bloc.

- **SubBytes** : il s'agit d'une opération de *substitution* qui remplace chaque octet de *State* par un autre octet choisi dans une table particulière (la « SBox ») déjà utilisée pour la fabrique des clefs de ronde.
- **ShiftRows** est une étape de *transposition* dans laquelle chaque élément de la matrice *State* est décalé de façon circulaire, vers la gauche, d'un certain nombre de colonnes : *c'est l'opération la plus simple de la ronde.*
- L'opération **MixColumns** effectue un *produit matriciel* dans \mathbb{F}_{256} de chaque colonne de 4 octets de la matrice *State*.
- Puis l'opération **AddRoundKey** combine à nouveau, par addition dans \mathbb{F}_{256} , chaque octet de la matrice *State* courante avec l'octet correspondant de la clef de ronde.

Enfin, une ronde finale est appliquée : elle correspond à une ronde partielle, dans laquelle l'opération **MixColumns** est omise.



Test de la fonction SubBytes

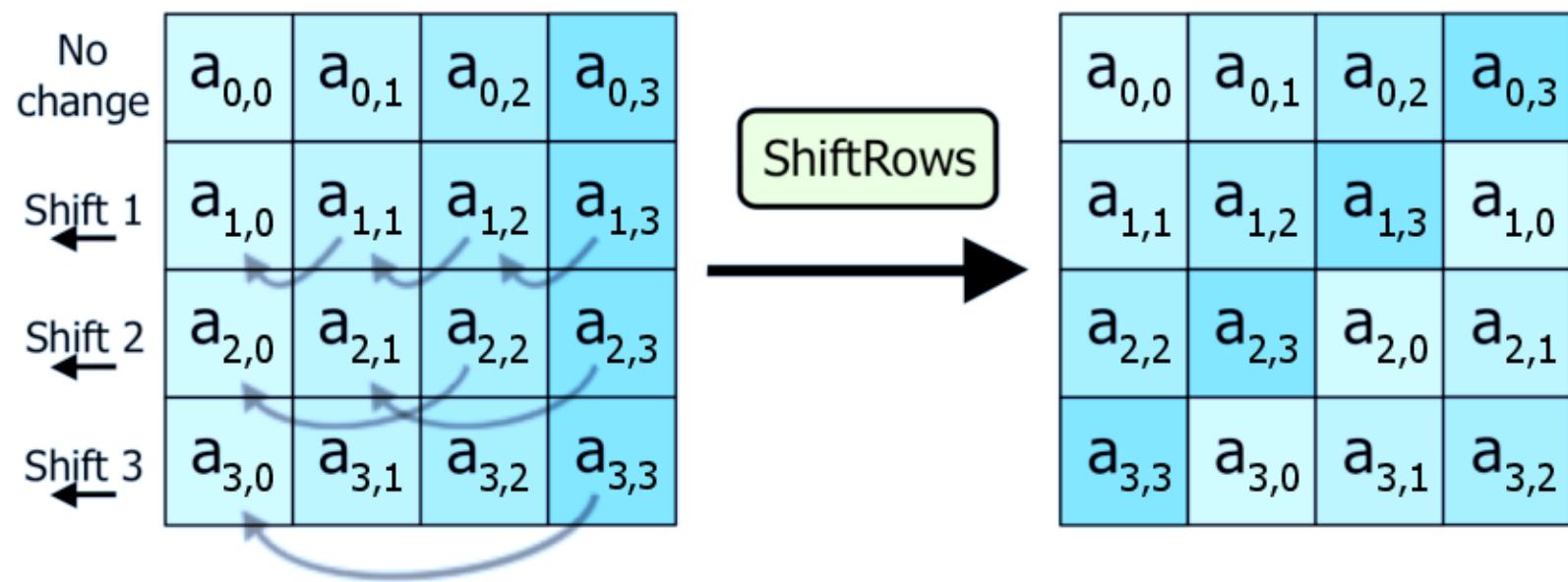
Test de SubBytes() :

Le bloc "State" en entrée vaut :

00	04	08	0C
01	05	09	0D
02	06	0A	0E
03	07	0B	0F

Le bloc "State" en sortie vaut :

63	F2	30	FE
7C	6B	01	D7
77	6F	67	AB
7B	C5	2B	76



Test de la fonction ShiftRows

Test de ShiftRows() :

Le bloc "State" en entrée vaut :

00 04 08 0C

01 05 09 0D

02 06 0A 0E

03 07 0B 0F

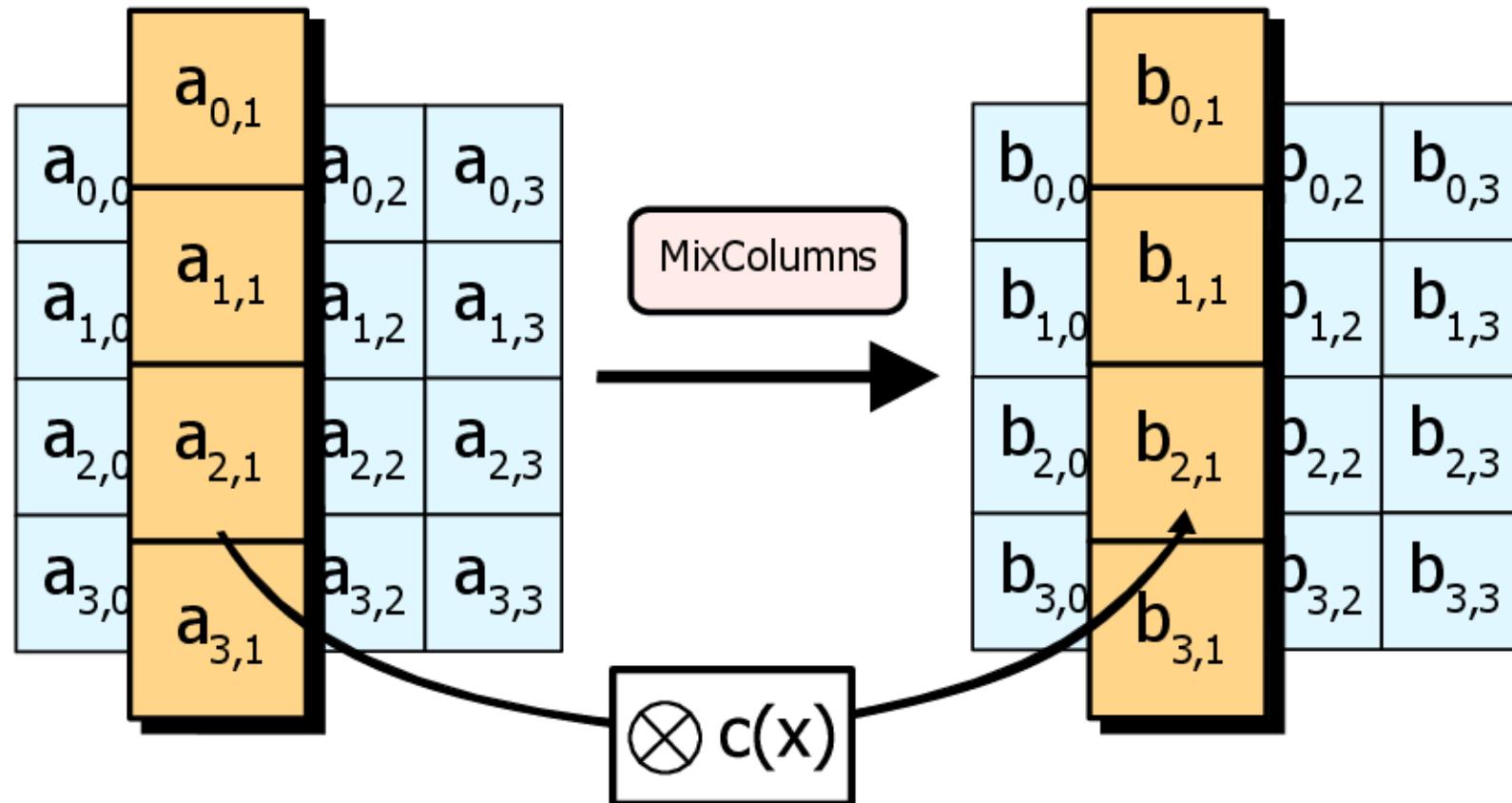
Le bloc "State" en sortie vaut :

00 04 08 0C

05 09 0D 01

0A 0E 02 06

0F 03 07 0B



L'opération MixColumns

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

Il s'agit d'une multiplication matricielle sur \mathbb{F}_{256} :

- $b_0 = 0x02 \times a_0 + 0x03 \times a_1 + 0x01 \times a_2 + 0x01 \times a_3$
- $b_1 = 0x01 \times a_0 + 0x02 \times a_1 + 0x03 \times a_2 + 0x01 \times a_3$
- $b_2 = 0x01 \times a_0 + 0x01 \times a_1 + 0x02 \times a_2 + 0x03 \times a_3$
- $b_3 = 0x03 \times a_0 + 0x01 \times a_1 + 0x01 \times a_2 + 0x02 \times a_3$

Le code de la multiplication adoptée

```
/* Fonction mystérieuse qui calcule le produit de deux octets */

uchar gmul(uchar a, uchar b) {
    uchar p = 0;
    uchar hi_bit_set;
    int i;
    for (i = 0; i < 8; i++) {
        if ((b & 1) == 1) p ^= a;
        hi_bit_set = (a & 0x80);
        a <<= 1;
        if (hi_bit_set == 0x80) a ^= 0x1b;
        b >>= 1;
    }
    return p;
}
```

Test de la fonction MixColumns

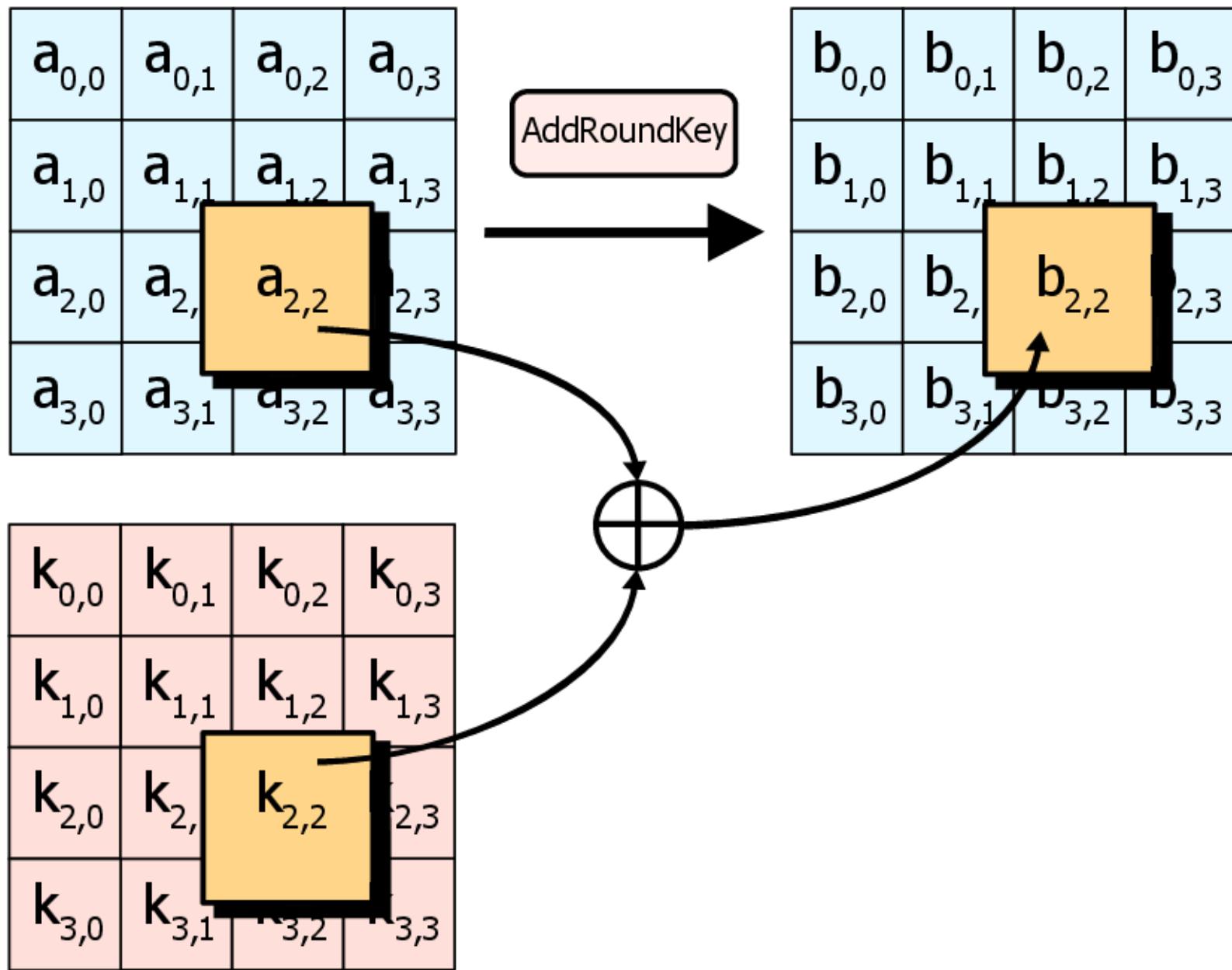
Test de MixColumns() :

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 la fonction AddRoundKey

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

Travaux pratiques

Il s'agira de calculer le chiffré du bloc nul avec la clef nulle.

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

Déchiffrement de l'AES

Le déchiffrement consiste simplement *à défaire les opérations effectuées lors du chiffrement dans l'ordre inverse* ; il faut donc pouvoir effectuer l'opération inverse de chaque transformation élémentaire utilisée lors du chiffrement.

Cela correspond, si l'on veut, à l'algorithme ci-dessous :

AES-ENCRYPT(State, K)

- 1 **KeyExpansion**(K, RoundKeys);
- 2 **AddRoundKey**(State, RoundKeys[N_r]);
- 3 **for** i \leftarrow N_r – 1 **to** 1
- 4 **do**
- 5 **InvShiftRows**(State);
- 6 **InvSubBytes**(State);
- 7 **AddRoundKey**(State, RoundKeys[i]);
- 8 **InvMixColumns**(State);
- 9 **InvShiftRows**(State);
- 10 **InvSubBytes**(State);
- 11 **AddRoundKey**(State, RoundKeys[0]);
- 12 **return**

Cryptographie symétrique par bloc

Master Informatique — Semestre 2 — UE optionnelle de 3 crédits

Chiffrements par blocs ou par flots

Dans les **schémas de chiffrement par blocs** chaque texte clair est découpé en blocs de même longueur et chiffré bloc par bloc.

La longueur des clés doit être suffisante pour que l'attaque exhaustive soit **irréaliste** (donc à nouveau : ≥ 128 bits).

Les schémas par blocs en général sont plus lents et nécessitent plus de moyens informatiques que les schémas par flots. Mais ils sont bien adaptés à la cryptographie civile, comme celle des banques.

La sécurité des schémas par blocs retenus comme standards (DES puis AES) est pragmatique : ces systèmes ont résisté aux tentatives d'attaques de toute la communauté cryptographique.

Le DES (Data Encryption Standard) qui date des années 70 a dû cependant être remplacé par l'AES (Advanced Encryption Standard), car sa clé de 56 bits est jugée de nos jours bien trop courte.



Bourrage

Nécessité du bourrage

Les algorithmes de chiffrement par blocs, tels que l’AES, supposent que la longueur l du fichier à chiffrer est un multiple de k octets, k étant la longueur fixée d’un bloc.

Pour de tels algorithmes, la phase de « bourrage » (ou « padding » en anglais) consiste à compléter le fichier à l’aide d’octets supplémentaires, avant de lancer le chiffrement proprement dit, afin de pouvoir découper le fichier en blocs de taille k .

Ces octets supplémentaires devront naturellement être supprimés lors du déchiffrement.

En général, les techniques de bourrages imposent d’ajouter des octets supplémentaires même si le taille du fichier est déjà un multiple de la taille du bloc.

Pourquoi ça ?

Bourrage PKCS#5

Le bourrage standard PKCS#5 (et PKCS#7) stipule qu'il faut, pour un fichier de taille l et des blocs de longueur k , ajouter $k - (l \bmod k)$ octets tous égaux et dont la valeur commune 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 $l \bmod k = k-1$

02 02 -- if $l \bmod k = k-2$

.

.

.

k k ... k k -- if $l \bmod k = 0$

✓ *Bourrage*

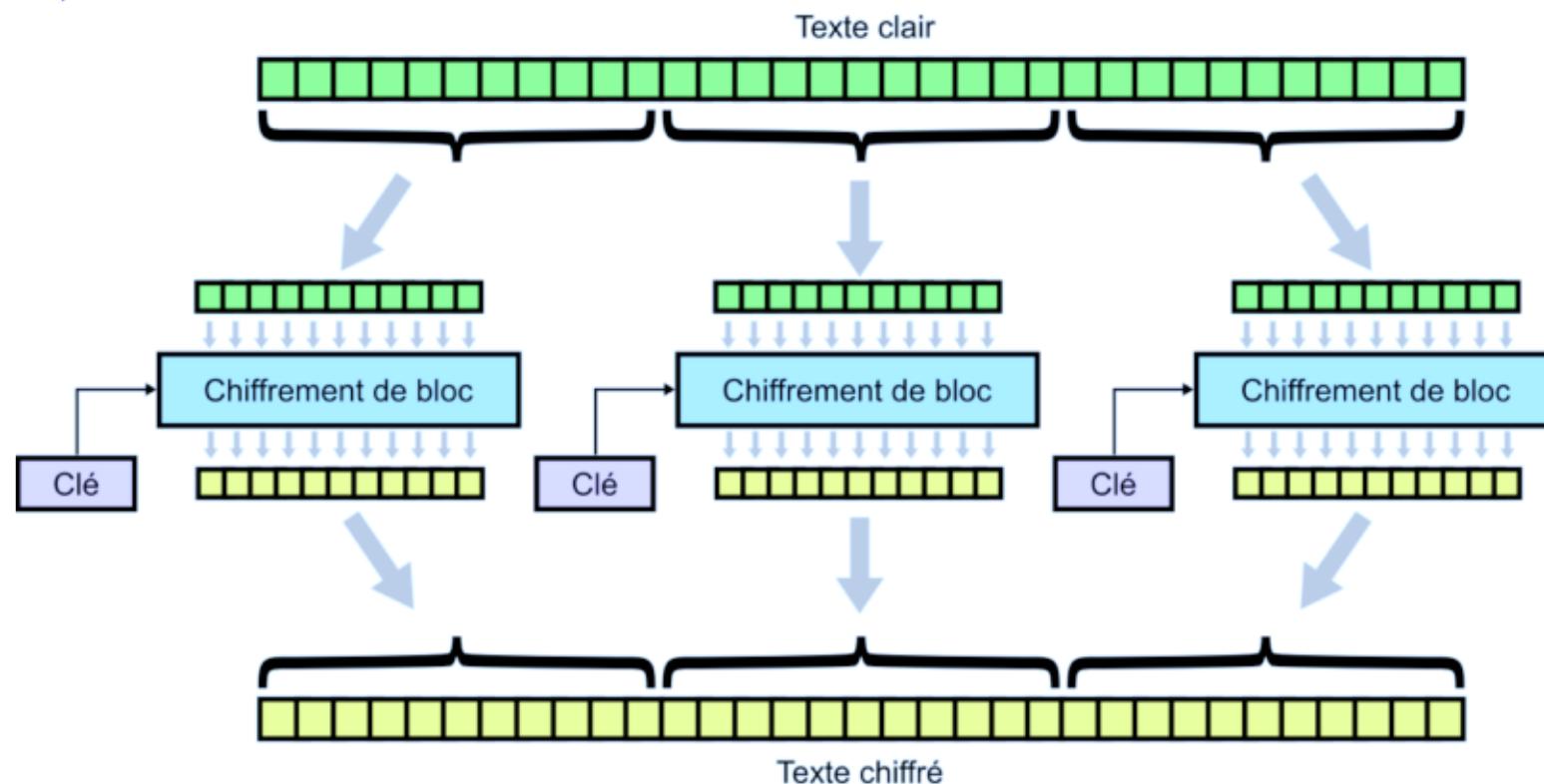
👉 *Modes opératoires*

Les modes opératoires

Le mode opératoire ECB (Electronic Code Book)

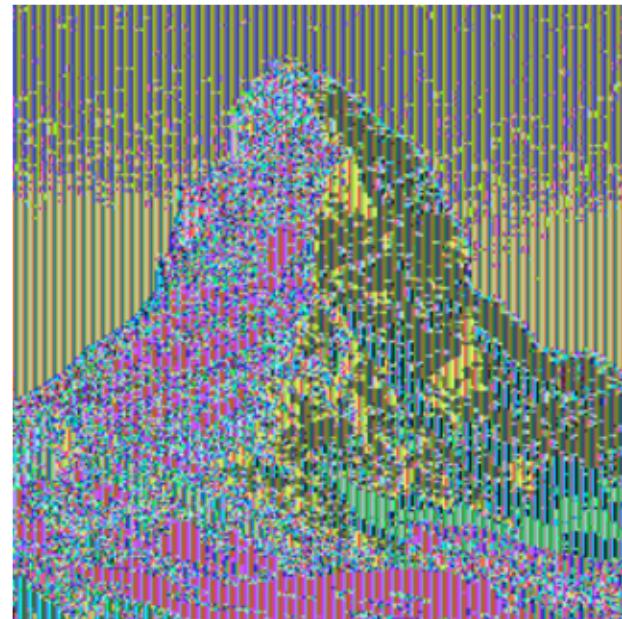
Il s'agit du mode le plus simple. Le message à chiffrer est subdivisé en plusieurs blocs qui sont chiffrés séparément les uns après les autres.

Le i -ème bloc du chiffré se déduit donc du i -ème bloc de clair par la relation
 $c_i = E_K(m_i)$



Le mode ECB

Deux blocs avec le même contenu seront chiffrés de la même manière : on peut donc tirer des informations à partir du texte chiffré en cherchant les blocs identiques.

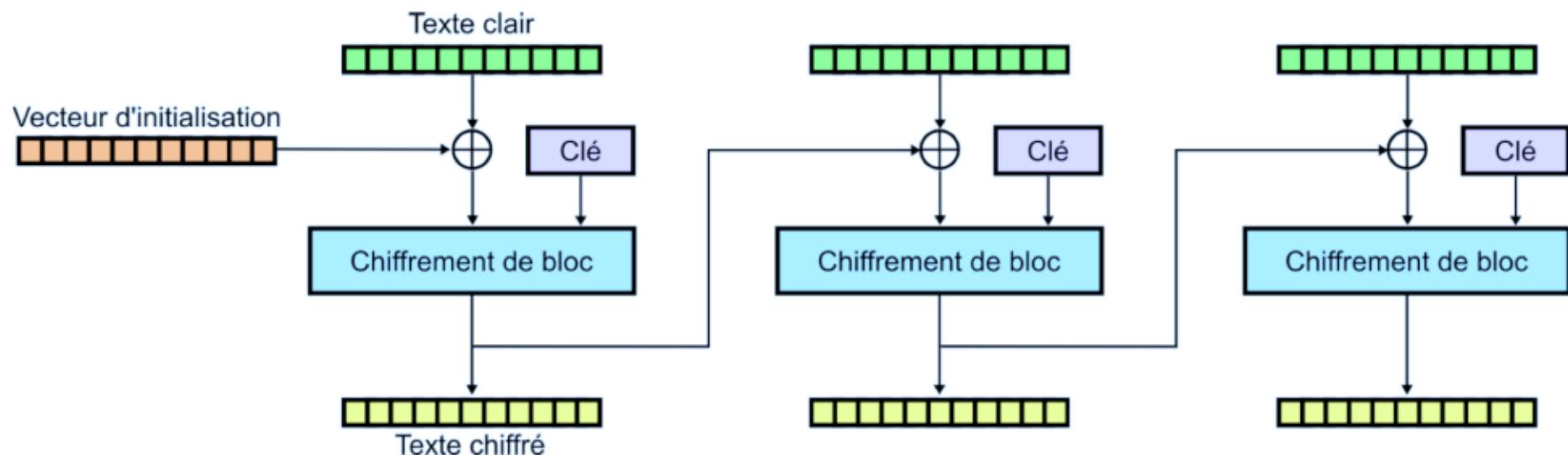


Le « dictionnaire de codes » établit les correspondances entre le clair et le chiffré, d'où le terme « codebook ».

Utilisation d'un vecteur d'initialisation

Le mode opératoire CBC (Cipher Block Chaining)

Dans ce mode $c_i = E_K(m_i \oplus c_{i-1})$ et $c_1 = E_K(m_1 \oplus IV)$ où IV est un **vecteur d'initialisation** choisi aléatoirement et *transmis (ou stocké) en clair avec le chiffré*.

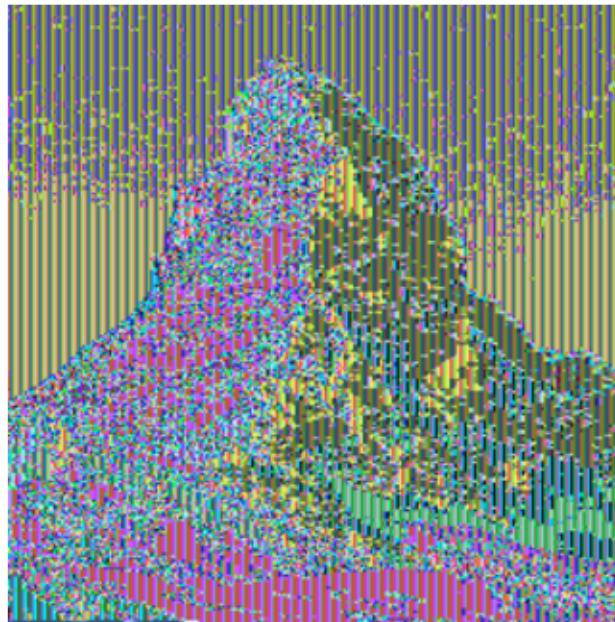


Comment s'opère le déchiffrement ?

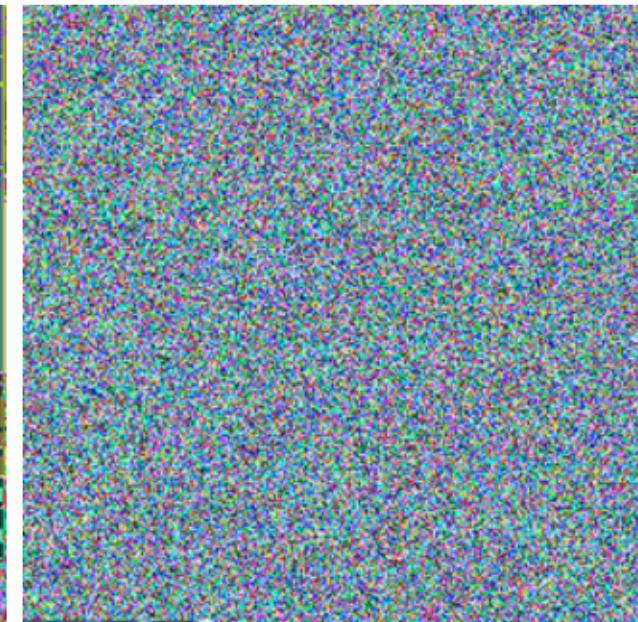
Le défaut du mode ECB corrigé par CBC sur une image



Clair



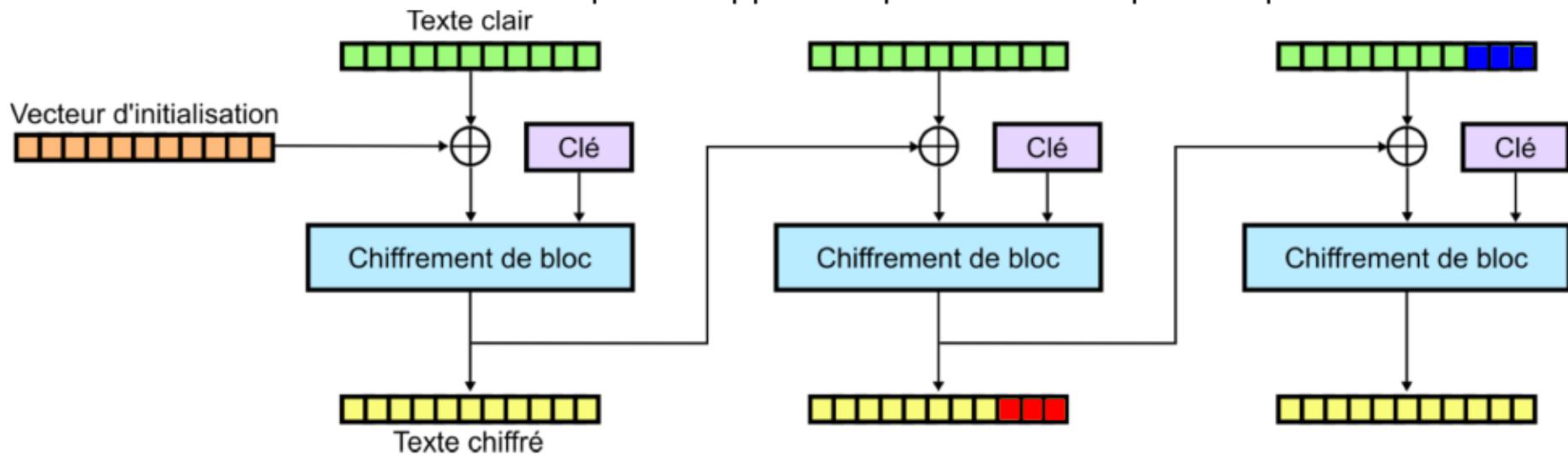
ECB



CBC

Mode « Cipher Text Stealing » (CTS)

Les modes CTS permettent d'éviter l'utilisation du **bourrage** dans les chiffrements par blocs ; ils produisent un message chiffré dont la taille est égale à celle du message clair. Ils sont très utilisés dans les protocoles et les formats de fichiers qui ne supportent pas une taille quelconque.

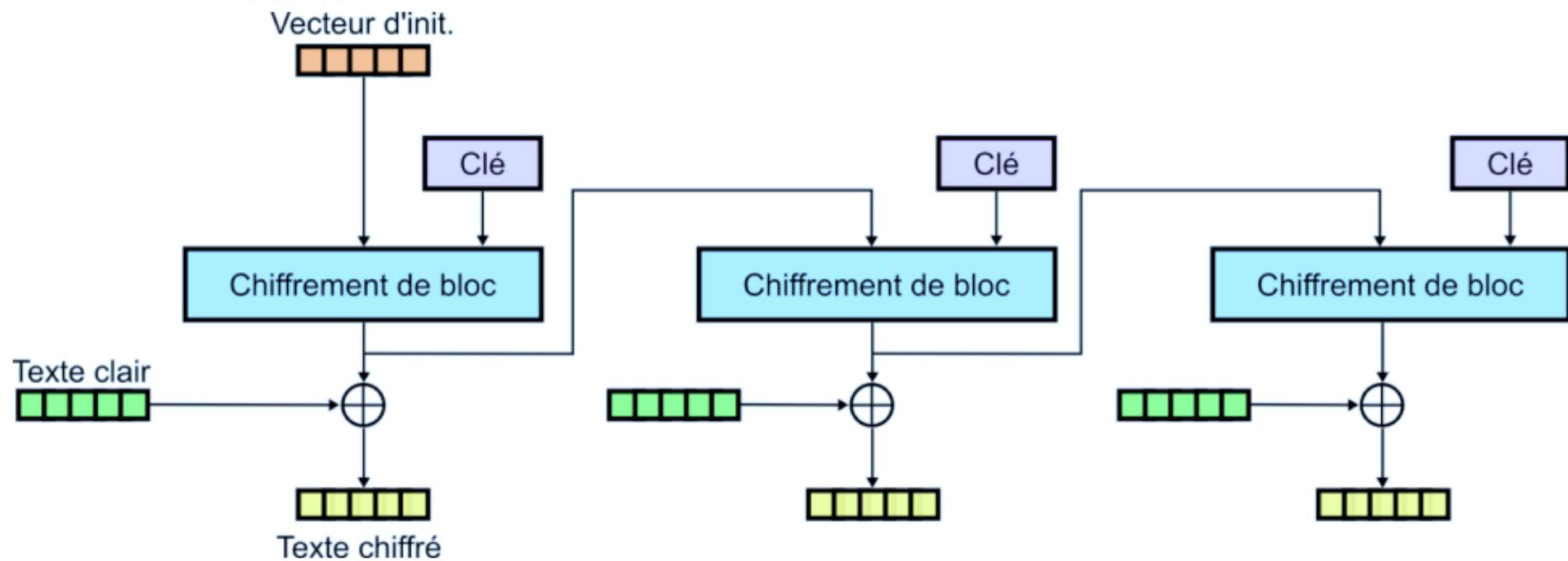


Dans ce mode, le dernier bloc est d'abord complété par des 0 et son chiffré est échangé avec l'avant-dernier chiffré ; de plus, les derniers octets de l'avant-dernier bloc chiffré sont **supprimés** !

Comment s'opère le déchiffrement ?

Autre mode : « Output Feedback » (OFB)

Ce mode et les modes suivants agissent **comme un chiffrement par flots** : ils génèrent un flux d'octets appliqués au texte clair : $c_i = m_i \oplus z_i$ avec $z_0 = \text{IV}$ et $z_i = E_K(z_{i-1})$.

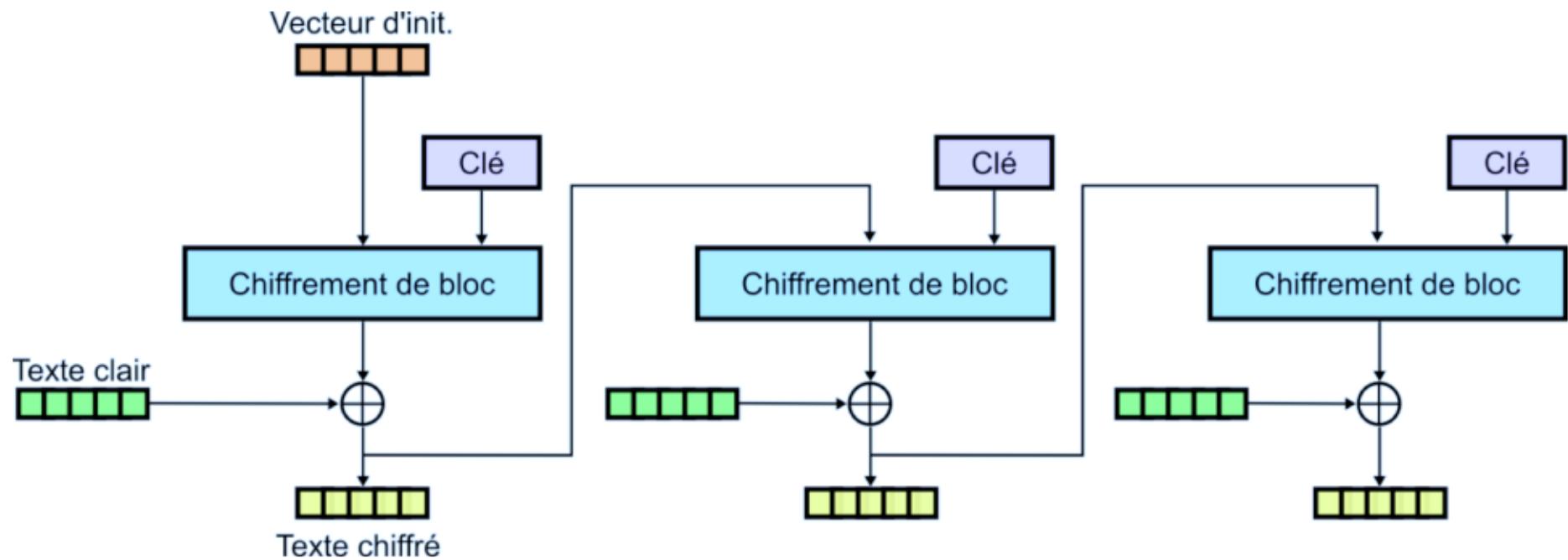


Sans surprise, ce mode est très fragile vis-à-vis d'une « attaque au clair ». Si l'attaquant connaît un message chiffré et le texte clair correspondant, il peut reconstituer aisément la suite d'octets utilisée.

Il ne faut jamais utiliser deux fois le même IV (avec la même clef).

Autre mode : « Cipher Feedback » (CFB)

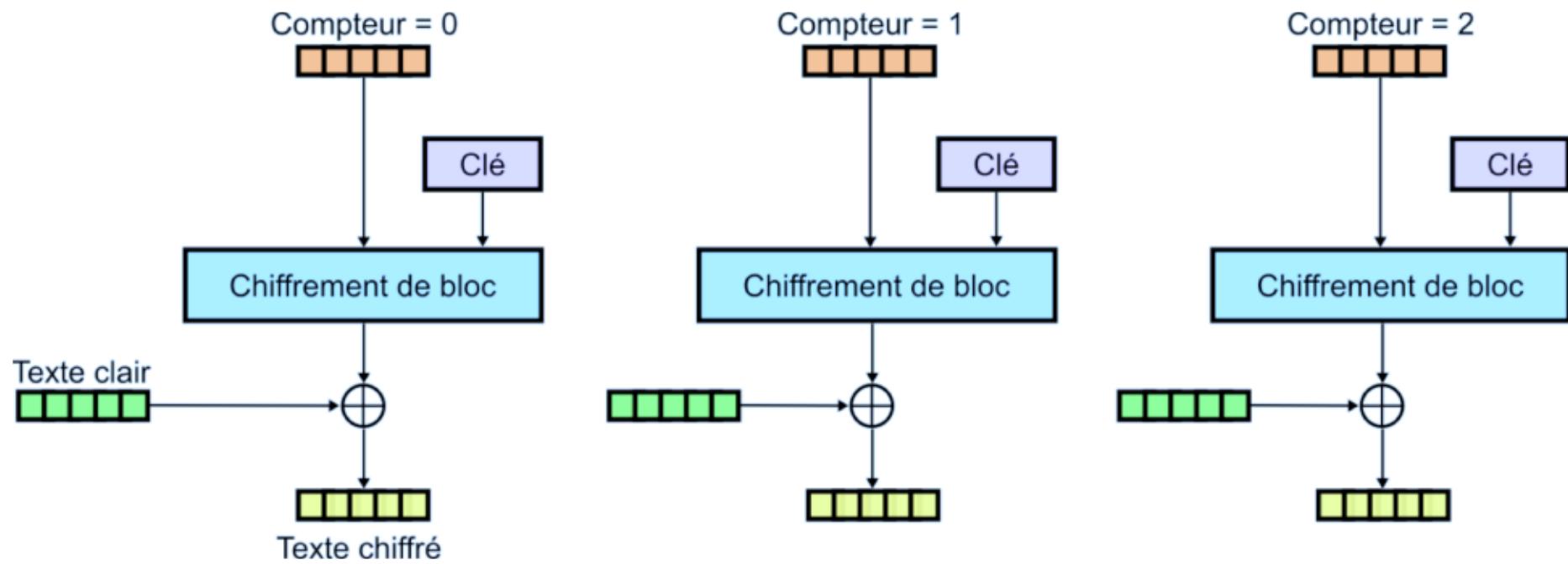
Dans ce mode le chiffré est obtenu par les formules $c_1 = m_1 \oplus E_K(IV)$ et $c_i = m_i \oplus E_K(c_{i-1})$.



Le flux d'octets pseudo-aléatoires dépend ainsi du message ; cependant, les éventuelles erreurs de calcul sur un bloc se propagent à l'ensemble du texte ultérieur.

Mode CTR

Dans le mode CTR, le flux d'octets de la clef longue est obtenu en chiffrant les valeurs successives d'un compteur : $c_i = m_i \oplus E_K(CPT)$ et



- Ce mode permet un *accès aléatoire aux données* ;
- Le chiffrement est *parallélisable*.

De plus, le compteur utilisé peut être lui aussi une suite *pseudo-aléatoire* qu'il sera facile de générer à partir de la graine. Celle-ci joue alors le rôle de IV.

- ✓ Bourrage
- ✓ Modes opératoires
- 👉 Le schéma de Feistel

Une méthode générale de chiffrement d'un bloc (1/2)

Dans un **schéma de Feistel**, le bloc à chiffrer est de longueur paire : il est découpé en une moitié gauche, L , et une moitié droite, R .

Il s'agit d'un système **itératif** (comme l'AES). Le i -ème tour du schéma prend en entrée un bloc (L_{i-1}, R_{i-1}) et le transforme, en faisant intervenir la i -ème sous-clé K_i en un bloc (L_i, R_i) .

Le premier tour prend en entrée le bloc (L, R) et le dernier tour produit en sortie le chiffré (L', R') .

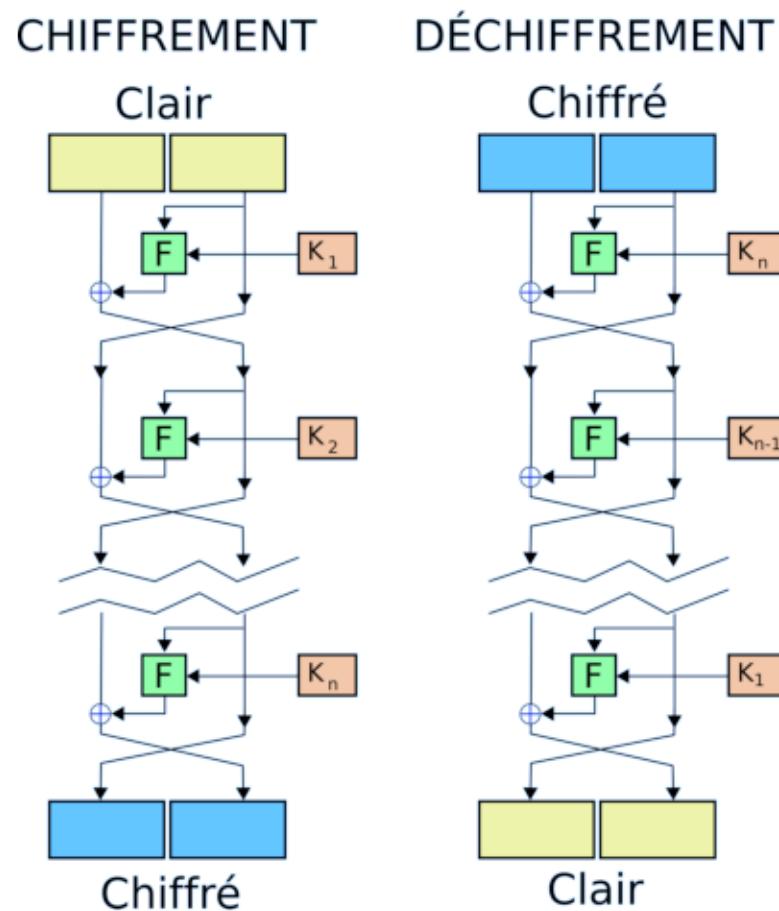
La relation entre (L_{i-1}, R_{i-1}) et (L_i, R_i) est la suivante :

$$L_i = R_{i-1} \text{ et } R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

où F est une **fonction de chiffrement** plus ou moins complexe.

Une méthode générale de chiffrement d'un bloc (2/2)

Ce chiffrement est **inversible**, que F soit une bijection ou non.



De plus, le message est déchiffré en suivant le même schéma dans *l'ordre inverse des clefs* et en *inversant les parties droite et gauche*.

Justification

Il suffit pour s'en convaincre d'observer que la première phase de déchiffrement annule la dernière phase de chiffrement. Autrement dit, de supposer qu'il n'y a qu'une phase.

Le message M se divise en deux parties : $M = M_1 \cdot M_2$

Le chiffré correspondant est $C = M_2 \cdot (M_1 \oplus F(M_2, K)) = C_1 \cdot C_2$.

Le message est déchiffré vaut

$$\begin{aligned} M' &= C_1 \cdot (C_2 \oplus F(C_1, K)) \\ &= C_1 \cdot ((M_1 \oplus F(M_2, K)) \oplus F(C_1, K)) \\ &\quad \text{car } C_2 = (M_1 \oplus F(M_2, K)) \\ &= M_2 \cdot ((M_1 \oplus F(M_2, K)) \oplus F(M_2, K)) \\ &= M_2 \cdot M_1 \rightsquigarrow M \end{aligned}$$

Ce qu'il faut retenir

- ① Les clefs utilisées dans un chiffrement symétrique sont essentiellement des suites d'octets choisis de manière aléatoire, et dont la longueur vaut **au moins 128 bits** ;
- ② Il existe deux grandes familles de chiffrements symétriques : par bloc et par flot ;
- ③ Le **chiffrement par flot** applique sur le texte clair un XOR avec une clef longue (dite « pseudo-aléatoire »), de la même longueur que le texte clair et générée à partir d'une clef courte ;
- ④ Les **chiffrements par bloc** utilisent une série d'opérations de diffusion et de confusion sur un bloc de taille fixée et sont en général renforcés par un **mode opératoire**, tel que CBC.
- ⑤ La phase de **bourrage** permet de s'assurer que le fichier se découpe parfaitement en blocs ; elle doit être annulée lors du déchiffrement.