Introduction à la cryptologie



Chapitre 2

Cryptographie symétrique



Pablo Rauzy <pr@up8.edu>
pablorauzy.fr/teaching/crypto

Cryptographie symétrique

Introduction

- La cryptographie *symétrique* (aussi appelée *à clef secrète*) est la plus ancienne forme de chiffrement.
- Le principe est d'avoir une clef partagée par l'émetteur et le destinataire, qui permet à la fois de chiffrer et déchiffrer les messages.

- Les chiffrements par substitutions que l'on a vu pendant la dernière séance (César, Vigenère) font parti de la cryptographie symétrique.
- Les techniques de chiffrement symétrique ont évoluées depuis.

Les techniques modernes

- Aujourd'hui on a besoin de chiffrer plus que de simples messages textes.
- Les algorithmes fonctionnent donc sur des bits, et non plus sur des lettres.
- D'autre part, la discipline s'est formalisée.

Deux grandes catégories

- Il existe deux grandes familles de chiffrements symétriques modernes :
  - les chiffrements par bloc, et
  - les chiffrements par flot.

Chiffrement par bloc

- Un algorithme de *chiffrement par bloc* ne sait chiffrer qu'un bloc de bits d'une taille donnée.
- Le principe est donc de découper le message en blocs de cette taille et de les chiffrer un par un, indépendemment.
- Exemples :
  - DES (ancien standard)
  - AES (standard actuel)

Chiffrement par flot

- Un algorithme de *chiffrement par flot* peut traiter des données de taille quelconque.
- Exemples :
  - A5/1 (GSM)
  - E0 (Bluetooth)
  - RC4 (WEP)

Masque jetable

- Avant de continuer sur les chiffrements par bloc et par flot, voyons ce qu'on appelle le *masque jetable* (ou *chiffrement de Vernam*).
- Le principe est celui d'un chiffrement de Vigénère, mais avec une clef aussi longue que le message (en pratique on utilise des bits et un ou-exclusif plutôt qu'une addition modulaire sur l'alphabet).
- Ce type de chiffrement est *incassable*, à condition que :
  - la clef soit bien aussi longue que le message à chiffrer,
  - les caractères (ou bits) composant la clef soient choisis aléatoirement,
  - chaque clef ne soit utilisée qu'une seule fois (d'où le "jetable").

Masque jetable

- Avant de continuer sur les chiffrements par bloc et par flot, voyons ce qu'on appelle le *masque jetable* (ou *chiffrement de Vernam*).
- Le principe est celui d'un chiffrement de Vigénère, mais avec une clef aussi longue que le message (en pratique on utilise des bits et un ou-exclusif plutôt qu'une addition modulaire sur l'alphabet).
- Ce type de chiffrement est *incassable*, à condition que :
  - la clef soit bien aussi longue que le message à chiffrer,
  - les caractères (ou bits) composant la clef soient choisis aléatoirement,
  - chaque clef ne soit utilisée qu'une seule fois (d'où le "jetable").
- En effet, si on ne connaît que le texte chiffré, et que toutes les clefs sont équiprobables, alors tous les textes clairs de cette longueur sont possibles avec la même probabilité.
- Cette sécurité est inconditionnelle (elle ne repose pas sur une difficulté de calcul).

En pratique

- En pratique, il est presque impossible de mettre ce type de chiffrement en application.
- Il est très difficile de générer des clefs parfaitement aléatoires.
- La distribution des clefs est fortement problématique (peut-être que la cryptographie quantique sera une solution).

Chiffrement par flot

- Le *chiffrement par flot* permets de chiffrer un flux continue de données.
- Il tente pour cela de reproduire le principe du masque jetable, mais en construisant algorithmiquement un *flux de clef* à partir d'une clef secrète.

Générateur de nombres pseudo-aléatoires

- Il faut donc une méthode permettant de générer un flux de clef composé de bits “aléatoires”, mais reproductible à partir de la clef secrète pour permettre le déchiffrement.
- La solution est l’utilisation d’un *générateur de nombres pseudo-aléatoires* (PRNG).
- Un PRNG est un algorithme qui génère une suite de nombre présentant certaines propriété du hasard (comme sembler indépendant les uns des autres).

Générateur congruentiel linéaire

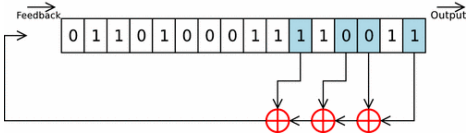
- Un *générateur congruentiel linéaire* est un algorithme de la forme :
  $x_{n+1} = (a \cdot x_n + b) \bmod m$.
- On appelle le terme initial x_0 la *graine*.
- Le choix des valeurs de a, b, et m est crucial.
- Mais quelles qu'elles soient, la suite est nécessairement ultimement périodique.

Générateur congruentiel linéaire

- Un *générateur congruentiel linéaire* est un algorithme de la forme :
  $x_{n+1} = (a \cdot x_n + b) \bmod m$.
- On appelle le terme initial x_0 la *graine*.
- Le choix des valeurs de a, b, et m est crucial.
- Mais quelles qu'elles soient, la suite est nécessairement ultimement périodique.
  - En effet elle ne peut prendre au maximum qu'un nombre fini de valeurs (égal à m).

Registre à décalage à rétroaction linéaire

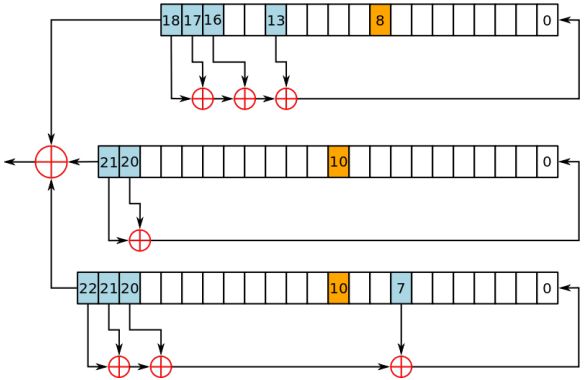
- Un *registre à décalage à rétroaction linéaire* (LFSR) produit une suite de bits.
- Le principe est d'opérer à chaque étape à un décalage d'un côté pour récupérer le bit "perdu", et de faire entrer par l'autre côté une combinaison linéaire (produite avec des ou-exclusif) des valeurs à cette étape de certains bits du registre.
- Pour un usage cryptographique, la fonction de rétroaction doit être choisi de façon à obtenir une période la plus grande possible.



- A5/1 est un exemple d'algorithme de chiffrement par flot.
- Il est utilisé pour les communications GSM en Europe et aux USA.
- Ce n'est pas un algorithme très sécurisé… par volonté.
  - A5/2 en est même une version affaiblie destinée à l'export.
  - On sait depuis les révélations de Snowden que la NSA sait casser A5/1 en temps réel.
- Il a été développé en 1987 mais a été gardé secret jusqu'à être retrouvé par rétro-ingénierie en 1999 (en partie en 1994) par M. Briceno.

Fonctionnement

- A5/1 fonctionne en utilisant 3 registres à décalage à rétroaction linéaire.



- Un registre est décalé si son bit orange correspond à la majorité des 3 bits orange.
- On insère alors un bit correspondant à un ou-exclusif entre ses bits en bleu.

Initialisation

- L'initialisation s'effectue avec la clef de 64 bits et le compteur initial de 22 bits :

1. Mettre tous les registres à 0.
2. Pour chacun des 64 bits k_i de la clef :
   - on effectue un ou-exclusif entre k_i et le lsb de chaque registre,
   - on décale chaque registre.
3. Pour chacun des 22 bits c_i du compteur :
   - on effectue un ou-exclusif entre c_i et le lsb de chaque registre,
   - on décale chaque registre.

Chiffrement par bloc

- Un algorithme de *chiffrement par bloc* chiffre un bloc de bits d'une taille donnée.

Modèle itératif

- La plupart des chiffrements par bloc fonctionnent suivant un *modèle itératif*.
- C'est à dire qu'une même transformation est appliquée plusieurs fois sur les données.
- On parle alors de *tours* (ou de *rondes*).
- Parfois, le premier et/ou le dernier tour sont différents des autres.
- Formellement, pour un algorithme avec des blocs de taille s et r tours :
  - On a une fonction F qui prend une clef k et le message m de s bits.
  - La fonction F est itérée r fois.
  - À chaque tour, on change la clef et on utilise le résultat de l'itération précédente :
    $c_1 = F(k_1, m)$
    $c_2 = F(k_2, c_1)$
    ...
    $c_r = F(k_r, c_{r-1})$
    $c = c_r$
  - Les *sous-clefs* k_i sont déduites de la clef k dans l'algorithme de *préparation des clefs*.

- La fonction F utilisée dans le modèle itératif doit être une permutation.

- La fonction F utilisée dans le modèle itératif doit être une permutation.
- C'est une permutation car c'est une bijection d'un ensemble dans lui même.

- La fonction F utilisée dans le modèle itératif doit être une permutation.
- C'est une permutation car c'est une bijection d'un ensemble dans lui même.
- C'est une bijection parce que c'est à la fois une injection et sur surjection.

- La fonction F utilisée dans le modèle itératif doit être une permutation.
- C'est une permutation car c'est une bijection d'un ensemble dans lui même.
- C'est une bijection parce que c'est à la fois une injection et sur surjection.
- C'est une injection parce qu'on veut pouvoir déchiffrer.

- La fonction F utilisée dans le modèle itératif doit être une permutation.
- C'est une permutation car c'est une bijection d'un ensemble dans lui même.
- C'est une bijection parce que c'est à la fois une injection et sur surjection.
- C'est une injection parce qu'on veut pouvoir déchiffrer.
- C'est une surjection pour des raisons de sécurité : on veut que la sortie dépende le moins possible de l'entrée, et idéalement que F soit une fonction pseudo-aléatoire, c'est à dire qu'on ne puisse pas distinguer ses sorties de celle d'une fonction aléatoire, et il faut donc a minima que toutes les valeurs de sortie soient possibles.

Préparation des clefs

- Il existe différentes façons de dériver des sous-clefs :
  - Simplement prendre des morceaux de la clef principale (TEA).
  - Lui faire subir des rotations variables par morceaux (DES).
  - Ou encore d'autres transformations plus complexes (PRESENT, AES).
- L'important et d'*équilibrer* l'usage des différents bits de la clef.

Confusion et diffusion

- La *confusion* correspond à la volonté de rendre la relation entre la clef de chiffrement et le texte chiffré la plus complexe possible.
  - Concrètement, on veut idéalement que chaque bit du chiffré dépendent de manière non-linéaire d'un maximum de bits du clair et d'un maximum de bits de la clef.
- La *diffusion* correspond à la volonté de décorréler les statistiques du texte clair avec celles du texte chiffré.
  - Concrètement, on veut que chaque bit du clair ou de la clef affecte un maximum de bits du chiffré.
- Ces concepts sont proposés par Shannon en 1949.



Claude Shannon (graffiti à *La Demeure du Chaos*)

Effet avalanche

- L'idée de l'*effet avalanche* est d'augmenter la diffusion au fur et à mesure que les données se propagent dans l'algorithme.
- Le *critère d'avalanche stricte* est vérifié si pour toute inversion d'un bit en entrée (donc sur le texte clair ou sur la clef), alors chaque bit en sortie (donc du chiffré) a une probabilité 0.5 d'être modifié.

Boîtes de substitution

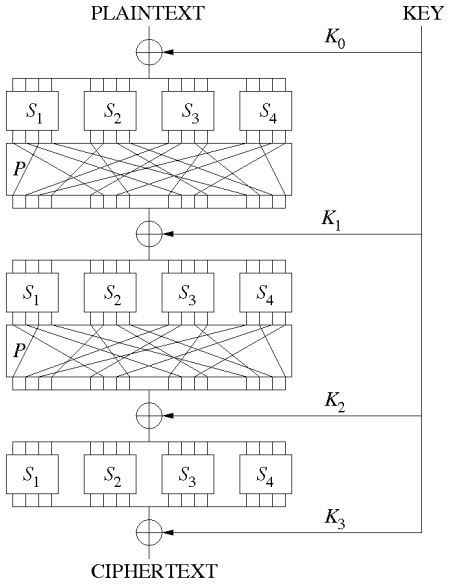
- Une *boîte S* prends m bits en entrée et produit n bits en sortie par une opération non linéaire (idéalement, une *fonction booléenne courbe*).
  - Idéalement, changer un bit en entrée d'une boîte S doit changer la moitié des bits en sortie.
  - De plus, chaque bit de sortie doit dépendre de tous les bits en entrée.
- On les implémente généralement avec une *table de correspondance*.
- L'utilisation de boîtes S contribue à la confusion.

Boîtes de permutation

- Une *boîte P* applique une permutation des bits du bloc.
- L'utilisation de boîtes P contribue à la diffusion.

Réseau de substitution-permutation

- Seule, une boîte S ou une boîte P n'a pas de robustesse cryptographique, mais en les combinant *de la bonne manière* on arrive satisfaire les propriétés de confusion et diffusion.
- On appelle *réseau de substitution-permutation* (RSP) une série de transformations d'un bloc de bits.
- Il s'agit généralement d'une suite de couches de plusieurs boites S suivi d'une boîte P.
- Entre les couches la clef de chiffrement est généralement ajoutée (le plus souvent sous la forme de *sous-clefs*, à l'aide d'un ou-exclusif).



- PRESENT est un exemple d'algorithme de chiffrement sur le modèle itératif qui a une structure de RSP.
- Il est publié par A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, et C. Vikkelsoe à CHES en 2007.
- C'est un algorithme léger, simple, et optimisé pour l'embarqué :
  - La taille des blocs est de 64 bits.
  - La clef fait 80 (ou 128) bits.
  - Il fait 31 tours similaires puis un ajout de clef final.
  - Il utilise une seule boîte S de 4×4 bits optimisée pour le hardware (14 portes logiques).
- La structure haut-niveau de PRESENT est la suivante :

```
1 k = generate_round_keys(key)
2 state = plaintext
3 for i = 1 to 31 do
4   state = add_round_key(state, k[i])
5   state = sbox_layer(state)
6   state = pbox_layer(state)
7 end
8 ciphertext = add_round_key(state, k[32])
```

Boîte S

- La boîte S de PRESENT est la suivante :

n	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(n)$	c	5	6	b	9	0	a	d	3	e	f	8	4	7	1	2

- C'est a priori la plus petite boîte S 4×4 satisfaisant les propriétés de sécurité (fonction courbe) en matériel.

Boîte P

- La permutation de PRESENT est la suivante :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51

i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55

i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59

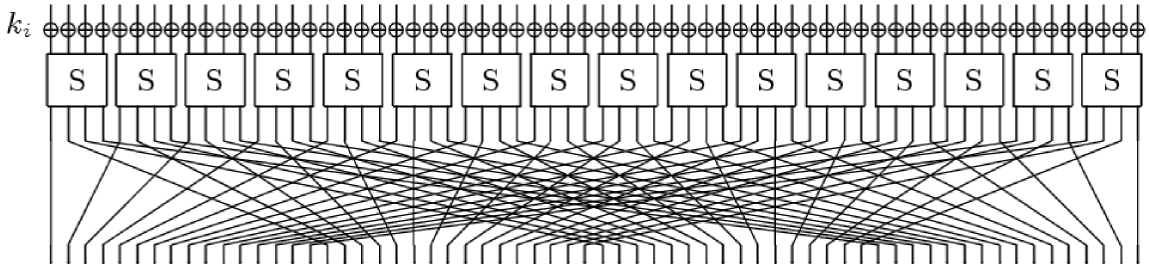
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

- C'est à dire $P(i) = \begin{cases} 16 \times i \mod 63, & 1 \leq i \leq 62 \\ i, & i \in \{0, 63\} \end{cases}$.

Préparation des clefs (80 bits)

- En voyant la clef k comme un tableau de 80 bits, la clef k_i du tour $1 \leq i \leq 32$ se calcule de la façon suivante :
  - $k'_1 = k$
  - $t_i = k'_{i-1}[19:80] + k'_{i-1}[0:19]$
  - $k'_i = t_i[0:15] + (t_i[16:20] \oplus i) + t_i[21:75] + S(t_i[76:80])$
  - $k_i = k'_i[0:64]$

- Représentation d'un tour de PRESENT en visualisation de RSP :



- Des versions affaiblies (avec moins de tour) de PRESENT ont été cassées dès 2007.
- La version complète l'a été plus récemment (2014, 2015) avec des techniques mathématiques avancées (biclique).

Réseau de Feistel

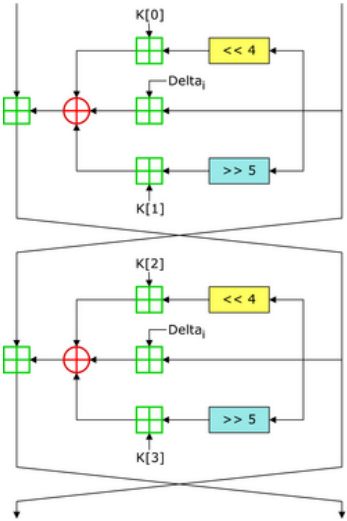
- Un *réseau de Feistel* est un autre type de structure d'algorithme de chiffrement par bloc sur le modèle itératif.
- Cette structure permet de construire une fonction F qui soit une permutation à partir de n'importe quelle fonction f.
- L'idée est de couper en 2 parties égales le message m :
  - $\overleftarrow{m}$ la partie gauche, et
  - $\overrightarrow{m}$ la partie droite.
- Ensuite, on peut calculer les c_i du modèle itératif de la manière suivante :
  - $c_0 = (\overleftarrow{c_0}, \overrightarrow{c_0}) = (\overleftarrow{m}, \overrightarrow{m})$,
  - $c_{i+1} = (\overrightarrow{c_i}, \overleftarrow{c_i} \oplus f(k_i, \overrightarrow{c_i}))$.
  - $c = c_r$ où r est le nombre de tours.
- De cette manière on peut déchiffrer même si f n'est pas une permutation :
  - On connaît c, donc on connaît $\overrightarrow{c_{r-1}} = \vec{c}$.
  - On peut calculer $f(k_{r-1}, \overrightarrow{c_{r-1}})$.
  - On a que $\overleftarrow{c_{r-1}} = \vec{c} \oplus f(k_{r-1}, \overrightarrow{c_{r-1}})$.
- La robustesse de F dépend directement ici de la robustesse de f (i.e., la difficulté de l'inverser sans connaître k).

- TEA est un exemple d'algorithme de chiffrement sur le modèle itératif qui a une structure de Feistel.
- Il est publié par D. Wheeler and R. Needham à FSE en 1994.
- C'est un algorithme connu pour la simplicité de son implémentation.
  - La taille des blocs est de 64 bits.
  - La clef fait 128 bits.
  - Il fait 64 tours similaires.
- La structure de haut niveau de TEA est la suivante :

```
1 cl, cr = message_halves(m)
2 k0, k1, k2, k3 = key_quarters(k)
3 for i = 0 to 31 do
4   cl = feistel_round(cr, k0, k1) // round 2*i
5   cr = feistel_round(cl, k2, k3) // round 2*i+1
6 end
```

Fonction de Feistel

- TEA a d'autres particularités que sa préparation de clef :
  - l'addition sur les entiers à la place du ou-exclusif pour mixer les deux parties du message,
  - idem pour ajouter les clefs de tours,
  - des décalages et une addition de constante magique (basé sur le nombre d'or) pour la diffusion,
  - des ou-exclusifs pour la confusion.



1 cycle de TEA
= 2 tours de Feistel

Le code complet

- L'algorithme TEA est suffisamment simple que son implémentation complète tienne sur un slide :

```
1  void TEA_encrypt (uint32_t v[2], uint32_t k[4]) {
2    uint32_t v0 = v[0], v1 = v[1];
3    uint32_t k0 = k[0], k1 = k[1], k2 = k[2], k3 = k[3];
4    uint32_t delta = 0x9e3779b9, sum = 0;
5    int i;
6
7    for (i = 0; i < 32; i++) {
8      sum += delta;
9      v0 += ((v1 << 4) + k0) ^ (v1 + sum) ^ ((v1 >> 5) + k1);
10     v1 += ((v0 << 4) + k2) ^ (v0 + sum) ^ ((v0 >> 5) + k3);
11   }
12
13   v[0] = v0;
14   v[1] = v1;
15 }
```

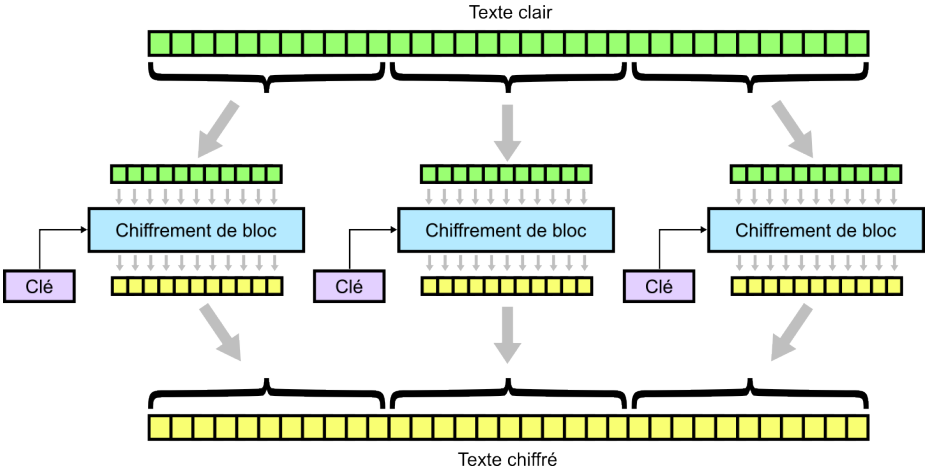
- TEA souffre de plusieurs faiblesses, il n'est a priori plus utilisé aujourd'hui (remplacé par XTEA puis XXTEA).
- Notamment, chaque clef possible est équivalente à trois autres.
  - Cela en fait un très mauvais algorithme de hachage !
  - Il a quand même été utilisé de cette manière pour la Xbox, et la faille a été utilisé pour Xbox Linux :

> *After reading Bruce Schneier's book on crypto, we learned that TEA was a really bad choice as a hash. The book says that TEA must never be used as a hash, because it is insecure if used this way. If you flip both bit 16 and 31 of a 32 bit word, the hash will be the same. We could easily patch a jump in the second bootloader so that it would not be recognized. This modified jump lead us directly into flash memory.*
>
> *But why did they make this mistake? Obviously the designers knew nothing about crypto - again! - and just added code without understanding it and without even reading the most basic books on the topic. A possible explanation why they chose TEA would be that they might have searched the internet for a "tiny" encryption algorithm - and got TEA.*

- Pour chiffrer tout un message avec un algorithme de chiffrement par bloc, on doit utiliser un *mode d'opération*.
- Ce mode défini comment sont traités les blocs consécutifs du message clair pour effectuer son chiffrement.
- Il existe plusieurs modes différents, et nous allons en voir quatre intéressants.

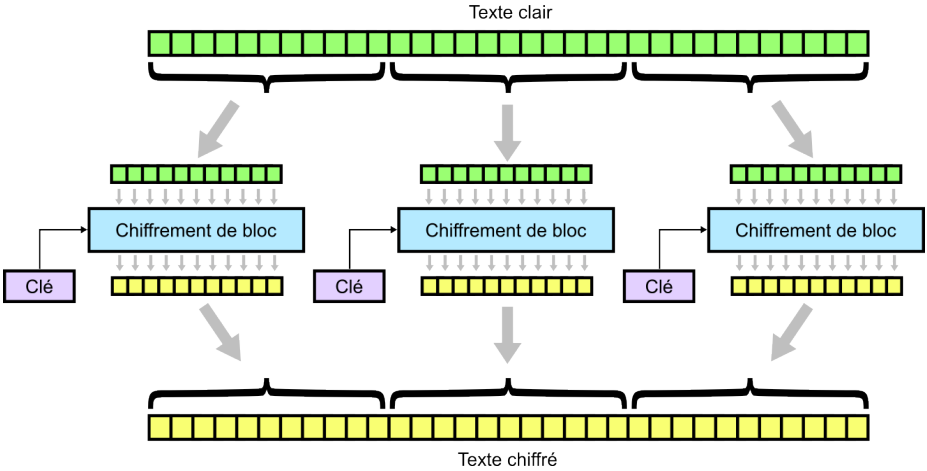
ECB – Electronic Code Book (dictionnaire)

- Le mode *ECB* est le plus simple : il s'agit simplement de découper le message et de chiffrer chaque bloc indépendemment.



ECB – Electronic Code Book (dictionnaire)

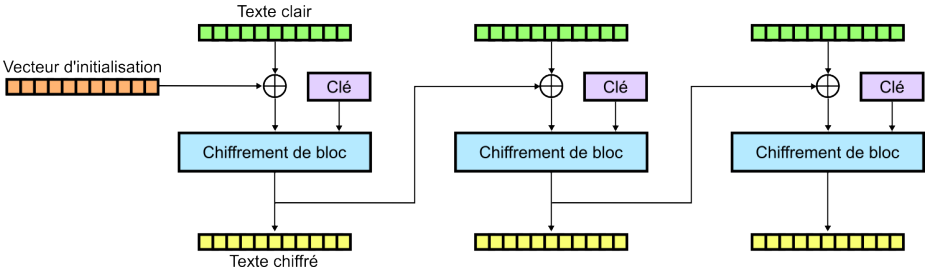
- Le mode *ECB* est le plus simple : il s'agit simplement de découper le message et de chiffrer chaque bloc indépendemment.



- Vulnérabilité : un même bloc clair aura toujours le même chiffré.

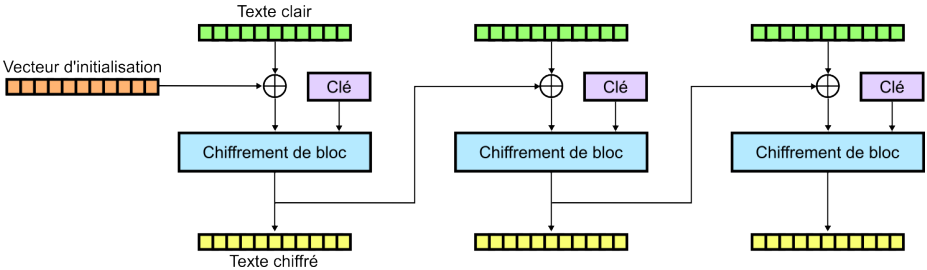
CBC – Cipher Block Chaining (enchaînement)

- Dans le mode *CBC*, on corrige la vulnérabilité de ECB en ajoutant (avec un ou-exclusif) le chiffré du bloc précédent au bloc clair.



CBC – Cipher Block Chaining (enchaînement)

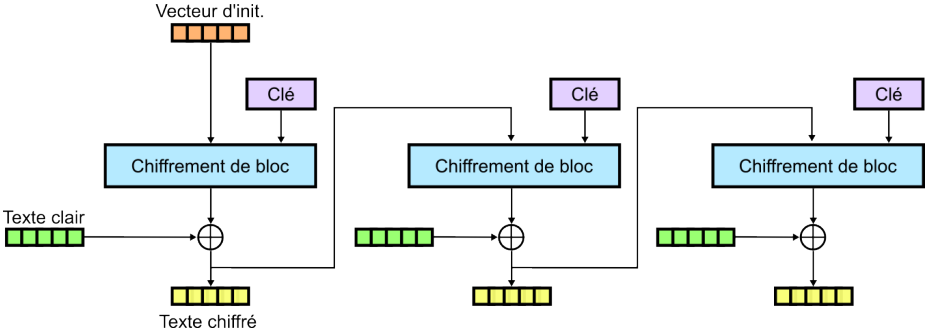
- Dans le mode *CBC*, on corrige la vulnérabilité de ECB en ajoutant (avec un ou-exclusif) le chiffré du bloc précédent au bloc clair.



- On ajoute un vecteur d'initialisation pour rendre chaque message unique.

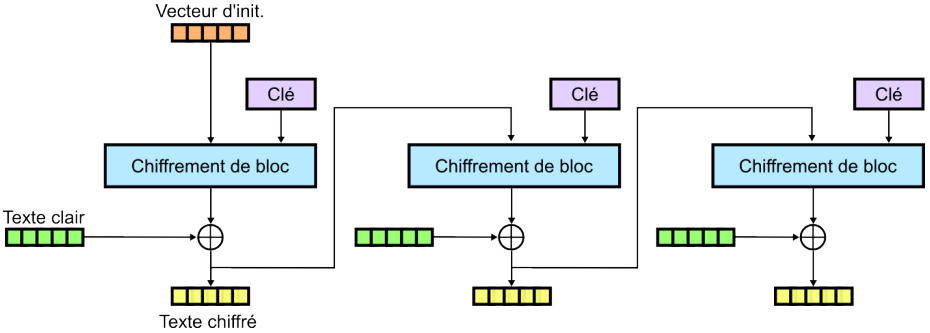
CFB – Cipher Feedback (rétroaction)

- Le mode *CFB* transforme un chiffrement par bloc en chiffrement par flot.
- Il fait cela en générant un flux de clef à partir du chiffrement du précédent bloc chiffré (ou d'un vecteur d'initialisation au début).



CFB – Cipher Feedback (rétroaction)

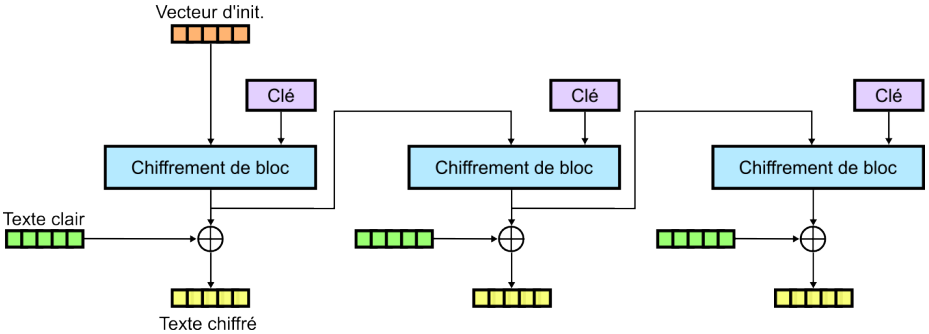
- Le mode *CFB* transforme un chiffrement par bloc en chiffrement par flot.
- Il fait cela en générant un flux de clef à partir du chiffrement du précédent bloc chiffré (ou d'un vecteur d'initialisation au début).



- Il faut avoir déjà chiffré le bloc précédent pour obtenir la suite du flux de clef.
- Il n'utilise que la fonction de chiffrement du chiffrement par bloc.

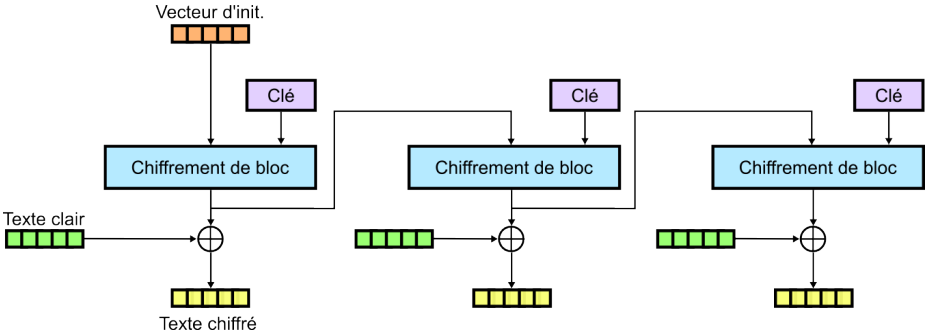
OFB – Output Feedback (rétroaction de sortie)

- Le mode *OFB* transforme aussi un chiffrement par bloc en chiffrement par flot.
- Cette fois-ci le flux de clef est généré uniquement à partir du vecteur d'initialisation : il ne va être sûr que si la fonction de chiffrement avec la clef choisie forment un bon générateur de nombres pseudo-aléatoires.



OFB – Output Feedback (rétroaction de sortie)

- Le mode *OFB* transforme aussi un chiffrement par bloc en chiffrement par flot.
- Cette fois-ci le flux de clef est généré uniquement à partir du vecteur d'initialisation : il ne va être sûr que si la fonction de chiffrement avec la clef choisie forment un bon générateur de nombres pseudo-aléatoires.



- Un attaquant à clair connu qui a le vecteur d'initialisation peut retrouver la clef.
- Il permet de pré-calculer le flux de clef.