

Table of contents

Cryptographie symétrique	1
Stream Ciphers (Cryptage en chaîne)	1
Introduction aux Stream Ciphers	1
Cryptage en chaîne (Stream Ciphers)	2
Stream Ciphers: Caractéristiques	3
Stream Ciphers Synchrones	3
Stream Ciphers Synchrones	5
Stream Ciphers Synchrones: Caractéristiques	5
Stream Ciphers Asynchrones	6
Stream Ciphers Asynchrones	7
Stream Ciphers Asynchrones: Caractéristiques	7
Générateurs de Keystreams : LSFR	8
Stream Ciphers: Générateurs de Keystreams	9
LSFRs: Quelques Remarques	10
RC4 : Stream Cipher Logiciel	10
Software Cipher Streams: RC4	14
RC4: Fonctionnement	14
Block Ciphers (Cryptage par Blocs)	15
1. Introduction aux Block Ciphers	15
2. Modes d'Opération des Block Ciphers	17
3. Product Ciphers et Feistel Ciphers	25
4. Data Encryption Standard (DES)	27
5. Triple-DES et Sécurité de DES	33
6. Advanced Encryption Standard (AES)	36
7. Attaques et Sécurité d'AES	42
8. Techniques de Cryptanalyse des Block Ciphers	45

Cryptographie symétrique

Stream Ciphers (Cryptage en chaîne)

Introduction aux Stream Ciphers

Définition et Principe

Les **stream ciphers** (chiffrements en flux) sont une famille de systèmes de cryptage caractérisés par :

- **Taille de bloc unitaire** : chaque bloc encrypté = 1 bit
- **Architecture en deux phases** :

1. **Génération du keystream** : production de la séquence de clés
2. **Substitution** : opération sur les bits du plaintext en fonction du keystream

Exemple classique : le *one-time pad*

- Génération : générateur (pseudo-)aléatoire
- Substitution : opération XOR (\oplus) avec le keystream

Caractéristiques Générales

Avantages :

- **Rapidité** : cryptage au niveau des registres, idéal pour le *streaming* en temps réel (vidéo)
- **Légèreté** : fonctionnent sur systèmes à ressources CPU limitées
- **Faible mémoire** : pas ou peu de buffering nécessaire
- **Erreurs non propagées** : retransmission des paquets défectueux suffisante (adapté aux transmissions sans fil - WiFi)

Inconvénients :

- **Dépendance à la qualité du keystream** : le caractère aléatoire (randomness) détermine la robustesse
- **Réutilisation dangereuse** : la réutilisation du keystream permet une cryptanalyse facile



Texte original

Cryptage en chaîne (Stream Ciphers)

- Les **stream ciphers** constituent une **famille de systèmes de cryptage** où la **taille du bloc encrypté est égale à 1 bit**.
- Les stream ciphers sont généralement composés de **deux phases**:
 - Une **phase de génération** de la séquence d'éléments formant la clé (le **keystream**).
 - Une **phase de substitution** où les bits du *plaintext* subissent une opération spécifique dépendante du keystream.
- Un exemple évident d'un stream cipher est le **one-time pad** avec:
 - Une phase de génération du keystream effectuée par un **générateur (pseudo-)aléatoire**.
 - Une phase de substitution qui consiste à effectuer un **xor** (\oplus) avec le keystream.

Stream Ciphers: Caractéristiques

- **Rapidité:** Le cryptage se fait directement au niveau des registres. Idéal pour des applications nécessitant un cryptage “*on the fly*” comme le **video streaming**.
- **Facilité:** Les opérations peuvent être effectuées par des systèmes ayant des ressources **CPU limitées**.
- **Pas (ou peu...) besoin de mémoire/buffering.**
- **Propagation des erreurs limitée ou absente:** la retransmission des paquets fautifs suffit normalement (adapté aux applications où les pertes de paquets sont fréquentes comme les **transmissions sans fil (WiFi)**).
- **Inconvénients:**
 - La **qualité en termes de randomness** du keystream généré détermine la **robustesse du système**.
 - La **réutilisation du keystream** permet une **cryptanalyse facile** (cf. le one-time pad).

Révision rapide

Stream Ciphers = cryptage bit par bit en 2 phases (génération keystream + substitution).

Avantages : rapides, légers, pas de propagation d’erreurs.

Inconvénients : qualité du keystream critique, réutilisation = vulnérabilité.

Stream Ciphers Synchrones

Principe de Fonctionnement

Dans un **stream cipher synchrone**, le keystream dépend **uniquement de la clé**, indépendamment du plaintext et du ciphertext.

Équations du processus :

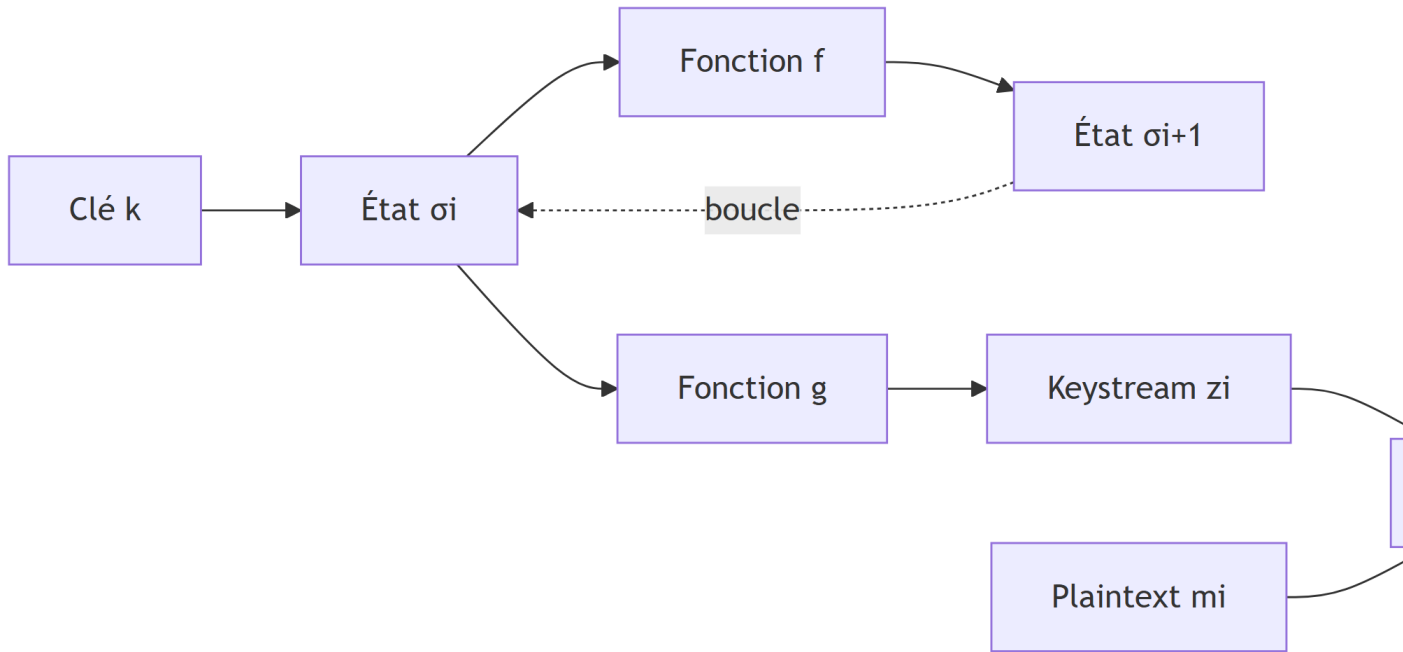
$$\sigma_{i+1} = f(\sigma_i, k)$$

$$z_i = g(\sigma_i, k)$$

$$c_i = h(z_i, m_i)$$

Où :

- σ_i : état à l'instant i (état initial σ_0 peut dépendre de k)
- k : clé secrète
- f : fonction de transition d'état
- g : fonction de production du keystream z_i
- h : fonction de sortie produisant le ciphertext c_i à partir du plaintext m_i



Caractéristiques

Exigence de synchronisation :

- Émetteur et récepteur doivent partager la même clé k **ET** le même état σ_i
- Perte de synchronisation = nécessité de mécanismes externes (marqueurs, analyse de redondance)

Propriétés :

- **Pas de propagation d'erreur** : modification du ciphertext n'affecte pas les séquences ultérieures
- **Attention** : suppression d'un ciphertext = désynchronisation du récepteur

Vulnérabilités aux attaques actives :

- Détection : insertion, élimination, replay de fragments

- Modification de bits : adversaire peut modifier des bits et analyser l'impact sur le plaintext
- **Solution** : mécanismes d'authentification supplémentaires nécessaires

Cas particulier : Stream Cipher Additif

Le cas le plus fréquent où :

- Fonctions f et g remplacées par un générateur aléatoire
- Fonction h = addition modulo 2 (XOR : \oplus)

Formule : $c_i = z_i \oplus m_i$

i Texte original

Stream Ciphers Synchrones

- Le **keystream** généré dépend seulement de la clé et non pas du plaintext ni du ciphertext.
- Le processus d'encryption d'un **stream cipher synchrone** est décrit par les équations suivantes:

$$\sigma_{i+1} = f(\sigma_i, k)$$

$$z_i = g(\sigma_i, k)$$

$$c_i = h(z_i, m_i)$$

avec σ_i l'état initial qui peut dépendre de la clé k , f la fonction qui détermine l'état suivant, g la fonction qui produit le keystream z_i et h la fonction de sortie qui produit le ciphertext c_i à partir du plaintext m_i .

Stream Ciphers Synchrones: Caractéristiques

- **Nécessitent la synchronisation** de l'émetteur et du récepteur: En plus d'utiliser la même clé k , les deux doivent se trouver dans le **même état** pour que le processus fonctionne. Si la synchronisation est perdue il faut des **mécanismes externes** pour la récupérer (marqueurs spéciaux, analyses de redondance du plaintext, etc.)
- **Pas de propagation d'erreur**. La modification du ciphertext pendant la transmission n'entraîne pas des perturbations dans des séquences de ciphertext ultérieures (cependant, la **suppression** d'un ciphertext provoquerait la **désynchronisation** du récepteur).
- **Attaques actives**: L'insertion, l'élimination ou le replay de parties de ciphertext sont **détectés** par le récepteur. Cependant, un adversaire pourrait **modifier certains bits** du ciphertext et analyser l'impact sur le plaintext correspondant. Des **mécanismes d'authentification d'origine** supplémentaires sont nécessaires afin de détecter ces attaques.

- **Cas les plus fréquent** des Stream Cipher Synchrones: le **stream cipher additif** (cf. le one-time pad) où les fonctions f et g générant le keystream sont remplacées par un **générateur aléatoire** et la fonction h est une **addition modulo 2 (xor)**.



Révision rapide

Synchrone : keystream = $f(\text{clé uniquement})$. Équations : $\sigma_{i+1} = f(\sigma_i, k)$, $z_i = g(\sigma_i, k)$, $c_i = h(z_i, m_i)$.

Exige synchronisation émetteur/récepteur. Pas de propagation d'erreur mais vulnérable aux modifications de bits.

Cas fréquent : cipher additif avec XOR.

Stream Ciphers Asynchrones

Principe de Fonctionnement

Aussi appelés **auto-synchronisés** (*self-synchronizing ciphers*).

Le keystream dépend de la clé **ET** d'un nombre fixe de ciphertexts précédents.

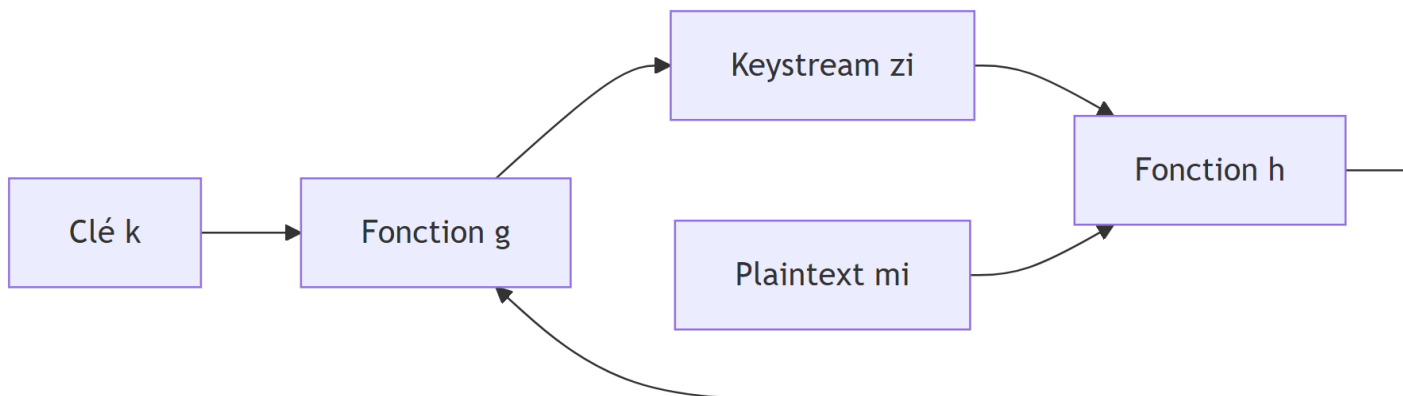
Équations du processus :

$$\sigma_i = (c_{i-t}, c_{i-t+1}, \dots, c_{i-1})$$

$$z_i = g(\sigma_i, k)$$

$$c_i = h(z_i, m_i)$$

Où σ_i représente un buffer des t derniers ciphertexts.



Caractéristiques

Auto-synchronisation :

- En cas d'insertion/élimination de ciphertexts, le récepteur se **re-synchronise automatiquement**
- Mécanisme : mémorisation (buffer) des derniers ciphertexts

Propagation d'erreurs limitée :

- Erreur se propage uniquement sur la **taille du buffer** (t bits)
- Après épuisement du buffer, décryption correcte reprend

Sécurité face aux attaques actives :

- **Meilleure détection** : modifications détectées grâce à la propagation d'erreurs
- **Attention** : l'auto-synchronisation permet au récepteur de continuer même après insertions/suppressions
- **Solution** : vérification de l'intégrité et l'authenticité du flux entier nécessaire

Diffusion des statistiques du plaintext :

- Chaque bit du plaintext influence **tous les ciphertexts subséquents**
- **Résultat** : meilleure dispersion des statistiques vs. cas synchrone
- **Application** : utiliser pour plaintexts à faible entropie ou fortement redondants

i Texte original

Stream Ciphers Asynchrones

- Aussi appelés **auto-synchronisés** (*self synchronizing ciphers*).
- Le **keystream** généré dépend de la clé ainsi que d'un nombre fixé de **ciphertexts précédents**.
- Le processus d'encryption d'un **stream cipher asynchrone** est décrit par les équations suivantes:

$$\sigma_i = (c_{i-t}, c_{i-t+1}, \dots, c_{i-1})$$

$$z_i = g(\sigma_i, k)$$

$$c_i = h(z_i, m_i)$$

avec σ_i , g et h comme pour le cas synchrone.

Stream Ciphers Asynchrones: Caractéristiques

- **Auto-synchronisation**: En cas d'élimination ou d'insertion de ciphertexts en cours de route, le récepteur est capable de **se re-synchroniser avec l'émetteur** grâce à la **mémorisation (buffer)** d'un nombre de ciphertext précédents.

- **Propagation d'erreurs limitée:** La propagation d'erreurs s'étend uniquement au **nombre de bits du ciphertext mémorisés** (taille du buffer). Après, la decryption se déroule à nouveau correctement.
- **Attaques actives:** La modification de fragments du ciphertext sera **plus facilement détecté** que dans le cas synchrone à cause de la propagation d'erreurs. Cependant, comme le récepteur est capable de s'auto-synchroniser avec l'émetteur, même si des ciphertexts sont éliminés ou insérés en cours de route, il convient de **vérifier l'intégrité et l'authenticité du flot entier**.
- **Diffusion des statistiques du plaintext:** Le fait que **chaque bit du plaintext aura une influence sur la totalité des ciphertexts subséquents** se traduit par une **plus grande dispersion des statistiques** du plaintext comparée au cas synchrone...
- ... Il convient, donc, d'utiliser des **stream ciphers asynchrones lorsque l'entropie des plaintexts est limitée** et pourrait permettre des attaques ciblées aux plaintexts fortement redondants.



Révision rapide

Asynchrone (auto-synchronisé) : $\text{keystream} = f(\text{clé} + \text{derniers ciphertexts})$. État σ_i = buffer de t ciphertexts précédents.

Auto-synchronisation automatique. Propagation d'erreur limitée au buffer.

Meilleure diffusion des statistiques → idéal pour plaintexts redondants/faible entropie.

Générateurs de Keystreams : LSFR

Contexte et Nécessité

Problématique : générer un keystream de longueur m à partir d'une clé secrète de longueur l avec $l \ll m$.

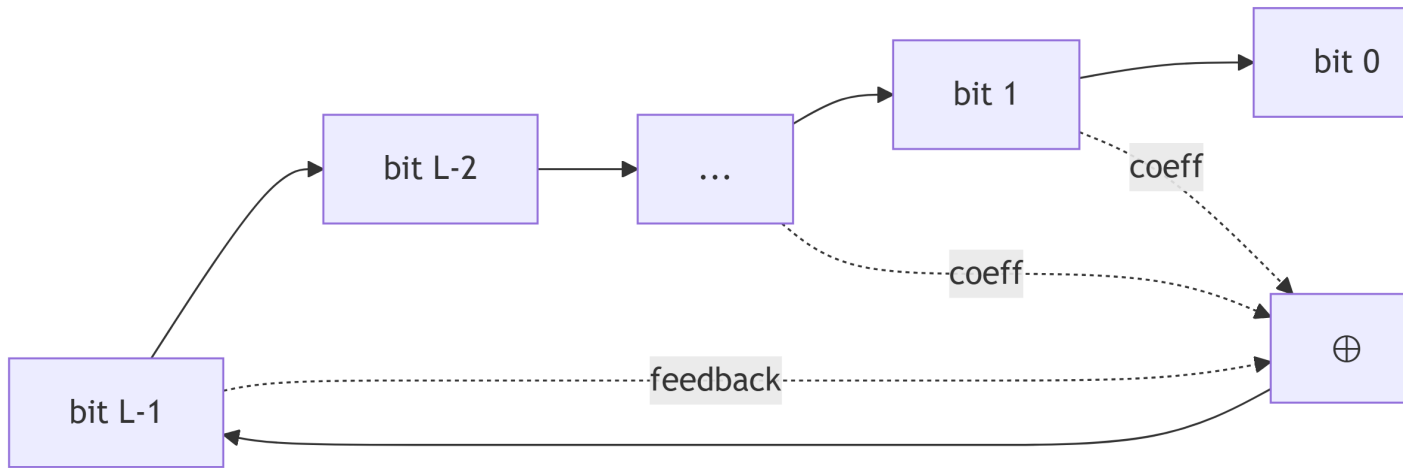
Solution : Linear Feedback Shift Register (**LSFR** ou **LFSR**)

Caractéristiques des LSFR

Avantages :

- **Implémentation hardware optimale** : circuits très efficaces
- **Périodes longues** : séquences de grande longueur
- **Bonne qualité aléatoire** : randomness notable
- **Base mathématique** : propriétés algébriques des combinaisons linéaires

Structure générique : LSFR de longueur L



Remarques Importantes sur les LSFR

Historique et Usage :

- Construction très répandue en cryptographie et théorie des codes
- Nombreux stream ciphers militaires basés sur LSFR

Limites de Sécurité :

- **Niveau de sécurité insuffisant** comparé aux block ciphers modernes
- **Vulnérabilité** : l'algorithme de Berlekamp-Massey permet de :
 - Déterminer la **complexité linéaire** d'un LSFR
 - Calculer un nombre arbitraire de séquences générées

Métrique : Complexité linéaire (*linear complexity*)

Solution d'Amélioration :

Remplacer la combinaison linéaire par une **fonction non linéaire** f

→ **Non Linear Feedback Shift Registers** (NLFSR)

i Texte original

Stream Ciphers: Générateurs de Keystreams

- Lorsqu'il convient de **générer un keystream d'une longueur m** à partir d'une **clé secrète de longueur l** avec $l \ll m$, on fait appel à des **générateurs de keystreams**.

- Le plus courant de ces générateurs est le **Linear Feedback Shift Register (LSFR)**.
- Un LSFR a les caractéristiques suivantes:
 - S'adapte **très bien aux implantations hardware**.
 - Produit des séquences de **périodes longues** et avec une **qualité aléatoire notable** (randomness assez forte)
 - Se base sur les **propriétés algébriques des combinaisons linéaires**.

LSFRs: Quelques Remarques

- Les LSFRs sont des constructions **très répandues** dans la cryptographie et dans la théorie de codes.
- Un **grand nombre de stream ciphers** basés sur les LSFRs (surtout dans la **sphère militaire**) ont été développés dans le passé.
- Malheureusement, le **niveau de sécurité offert par ces systèmes est jugé insuffisant** de nos jours (comparé à celui des blocks ciphers...)
- La **métrique** permettant d'analyser un LSFR est sa **complexité linéaire** (*linear complexity*). L'**algorithme de Berlekamp-Massey** permet de déterminer la complexité linéaire d'un LSFR et de calculer ainsi un nombre arbitrairement grand de séquences générées par un LSFR.
- Une solution pour **augmenter la complexité** est de substituer la combinaison linéaire des bits du ciphertext par une **fonction non linéaire f** . Ce sont les **Non Linear Feedback Shift Registers**.

Révision rapide

LSFR : générateur de keystream long (m) depuis clé courte (l). Base = combinaisons linéaires.

Avantages : hardware efficace, périodes longues.

Problème : sécurité insuffisante, vulnérable à Berlekamp-Massey (calcul de complexité linéaire).

Solution : NLFSR (fonction non linéaire).

RC4 : Stream Cipher Logiciel

Présentation Générale

RC4™ (*Rivest Cipher 4*) développé en 1987 par Ron Rivest pour RSA Security.

Caractéristiques principales :

- **Clé variable** : longueur flexible
- **Extrêmement rapide** : $10\times$ plus rapide que DES
- **Mode synchrone** : keystream indépendant du plaintext/ciphertext

Historique :

- 1987-1994 : breveté, détails confidentiels (contrat NDA requis)
- 1994 : publication non officielle dans un newsgroup
- Depuis : analyse intensive par la communauté cryptographique

Architecture

Composants clés :

- **S-box** : boîte de substitution 8×8 (256 entrées)
 - Contenu : permutation des nombres 0 à 255
 - Dépend de la clé principale de longueur variable : $0 < \text{len}(k) \leq 255$
- **Combinaisons** : linéaires et non linéaires
- **Chiffrement final** : XOR entre keystream et plaintext

Applications et Sécurité

Utilisations commerciales (nombreuses) :

- Lotus Notes
- Oracle SQL
- Microsoft Windows
- SSL/TLS
- Et bien d'autres...

Analyses et Vulnérabilités :

- Travaux exhaustifs sur le key scheduling et le PRGA
- **Faible majeure** : implémentation dans WEP (WiFi Wired Equivalent Privacy)
 - Protocole WEP complètement compromis
 - Problème : mode d'utilisation défaillant, pas l'algorithme RC4 lui-même

Fonctionnement

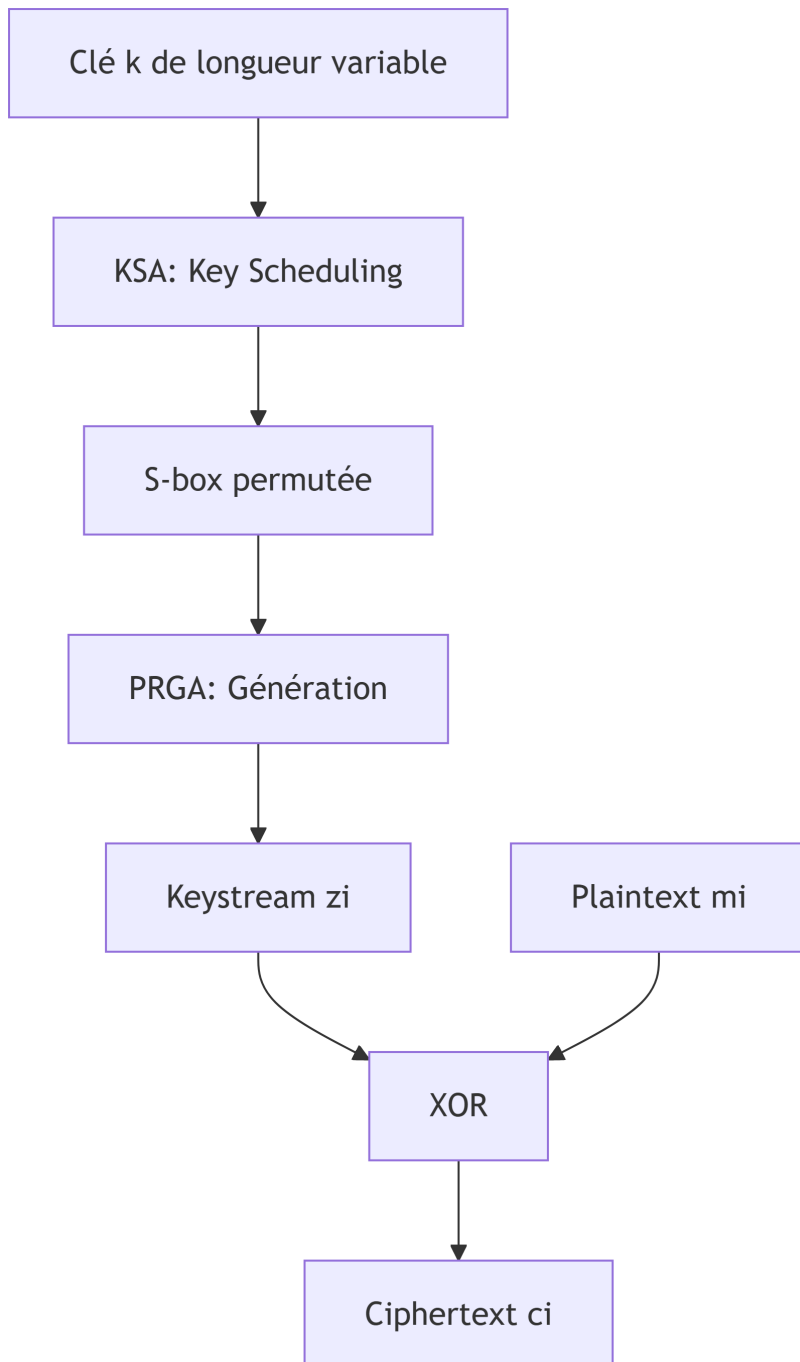
RC4 se décompose en **deux étapes** :

1. Key Scheduling Algorithm (KSA)

- Responsable de la permutation initiale de la S-box
- Fonction de la clé de longueur variable $len(k) = l$

2. Pseudo Random Generator Algorithm (PRGA)

- Génère le keystream de taille arbitraire
- S'appuie sur la S-box permutée par KSA



Software Cipher Streams: RC4

- Le grand **désavantage des stream ciphers basés sur des registres** est qu'ils sont **très lents en version programmée** dans une machine générique. **RC4TM** est un **stream cipher à clé variable** développé en **1987 par Ron Rivest** pour la société RSA security. Il est **très rapide (10 fois plus rapide que DES !)**
- Pendant 7 ans, cet algorithme était **breveté** et les détails son fonctionnement interne était dévoilés seulement après la **signature d'un contrat de confidentialité**. Depuis sa **publication (non officielle) dans un newsgroup en 1994**, il est globalement discuté et analysé dans toute la communauté cryptographique.
- L'algorithme travaille en **mode synchrone** (le keystream est indépendant du ciphertext et du plaintext).
- Il est composé de **combinaisons linéaires et non linéaires**. L'élément clé est une **boîte de substitution (S-box) de taille 8×8** dont les entrées sont une **permutation des chiffres 0 à 255**. La permutation est une **fonction de la clé principale** de taille variable avec $0 < len(k) \leq 255$. **L'encryption finale est obtenue par un xor** entre le keystream et le plaintext.
- RC4 est utilisé dans un **grand nombre d'applications commerciales**: Lotus Notes, Oracle SQL, MS Windows, SSL, etc. Il est l'objet d'un grand nombre de **travaux analytiques et exhaustifs** qui ont réussi à **compromettre la sécurité** du key scheduling et du PRGA.
- En particulier l'application de RC4 sur les **Wired Equivalent Privacy (WiFi WEP) protocole a été "cassée"** suite à une **faille dans le mode d'utilisation** du protocole.

RC4: Fonctionnement

- L'algorithme est constitué de **deux étapes**:
 - Le **Key Scheduling Algorithm (KSA)**: Responsable de la **permutation initiale** qui remplira la S-box en fonction de la clé de longueur variable $len(k) = l$.
 - Le **Pseudo Random Generator Algorithm (PRGA)**: Génère le **keystream de taille arbitraire** en s'appuyant sur la S-box.

Révision rapide

RC4 : stream cipher logiciel, clé variable, 10× plus rapide que DES.

Architecture : S-box 8×8 (permutation 0-255) + XOR.

2 étapes : KSA (permutation S-box) + PRGA (génération keystream). Mode synchrone.

Vulnérabilité : WEP cassé (faille d'utilisation). Utilisé dans SSL, Windows, Oracle...

Block Ciphers (Cryptage par Blocs)

1. Introduction aux Block Ciphers

Définition et Principe

Un **block cipher** (chiffrement par blocs) est une fonction cryptographique qui :

- **Transforme des blocs de taille fixe** : fait correspondre un bloc de n bits à un autre bloc de la même taille
- **Est paramétrisée par une clé** : la clé K de k bits définit la transformation
- **Doit être bijective** : pour permettre un décryptage unique
- **Chaque clé = bijection différente** : garantit la variabilité

Taille nominale : taille d'entrée du bloc sur lequel s'applique l'encryption

Critères de Qualité

1. Taille/Entropie de la clé

- Clés idéalement **équiprobables** avec entropie = k bits
- Forte entropie protège contre les **attaques brute-force**
- **Minimum requis** : 128 bits pour les block ciphers modernes

2. Performances

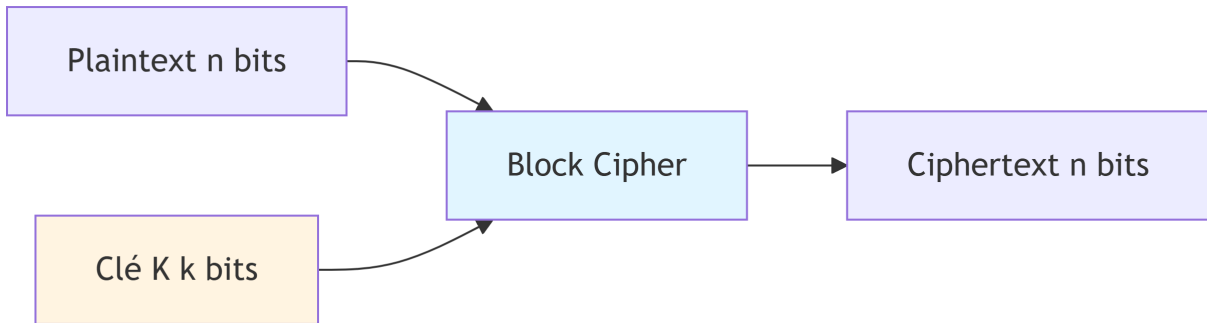
- Vitesse d'exécution
- Efficacité en software/hardware

3. Taille du bloc

- Bloc trop petit = vulnérabilité aux **dictionnaires plaintext/ciphertext**
- **Standard moderne** : blocs 128 bits

4. Résistance cryptographique

- Résistance aux techniques connues :
 - Cryptanalyse linéaire
 - Cryptanalyse différentielle
 - Meet in the middle
- **Effort de cryptanalyse** équivalent au brute force



i Texte original

Cryptage par Blocs (Block Ciphers)

- Les **block ciphers symétriques** constituent la **pierre angulaire de la cryptographie**. Leur fonctionnalité principale est la **confidentialité** mais ils sont également à la base des services d'**authentification**, **fonctions de hachage**, **génération aléatoire**, etc.
- **Définition:** Un block cipher est une **fonction** qui fait correspondre à un **bloc de n bits** un autre bloc de **la même taille**. La fonction est **paramétrée par une clé K de k bits**. Afin de permettre une **decryption unique**, la fonction doit être **bijective**. **Chaque clé définit une bijection différente**. La **taille d'entrée du bloc** sur lequel s'applique l'encryption s'appelle aussi **taille nominale de l'algorithme**.
- **Critères pour évaluer la qualité d'un block cipher:**
 - **Taille/Entropie de la clé:** Idéalement, les clés sont **équiprobables** et l'espace des clés a une **entropie égale à k** . Une **forte entropie** de la clé protège des **attaques brute-force** à partir de chosen/known plaintexts. Les block ciphers modernes doivent avoir des **clés d'au moins 128 bits**.
 - **Performances**
 - **Taille du bloc:** Un bloc **trop petit** permettrait des attaques où des “**dictionnaires**” **plaintext/ciphertext** seraient construits. De nos jours, des **blocs de taille 128 bits** deviennent courants.
 - **Résistance cryptographique:** Le block cipher doit se montrer **résistant** à des techniques de cryptanalyse connues: **cryptanalyse linéaire ou différentielle**, **meet in the middle**, etc. L'**effort inhérent** à ces attaques (complexité, stockage, parallélisation, etc.) doit être **équivalent à celui d'une attaque brute force**.

 Révision rapide

Block cipher : fonction bijective transformant blocs de n bits avec clé K de k bits.

Critères : entropie clé 128 bits, taille bloc 128 bits, résistance cryptanalyse = effort brute force. **Usage** : confidentialité, authentification, hachage, génération aléatoire.

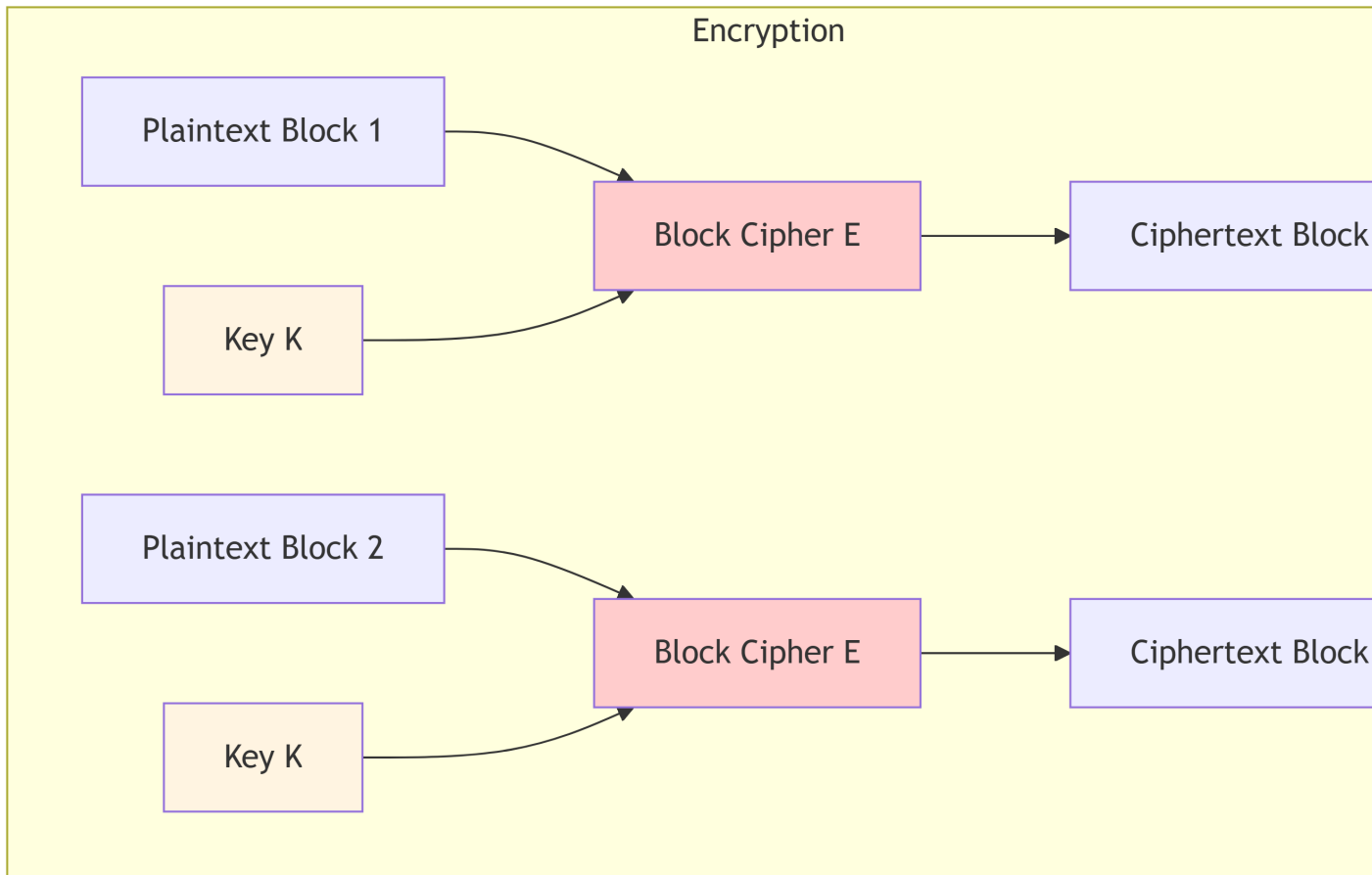
2. Modes d'Opération des Block Ciphers

2.1 Electronic Codebook (ECB)

Principe : chaque bloc de plaintext est encrypté **indépendamment** avec la même clé.

$$c_i = E_K(m_i)$$

$$m_i = D_K(c_i)$$



Caractéristiques :

- **Plaintexts identiques** → ciphertexts identiques (prévisible)
- **Pas de propagation d'erreurs** : erreur sur c_j n'affecte que m_j
- **Patterns visibles** : structure du plaintext transparente dans le ciphertext
- **Parallélisable** : chaque bloc traité indépendamment

Vulnérabilité majeure : Ne doit PAS être utilisé pour des données redondantes

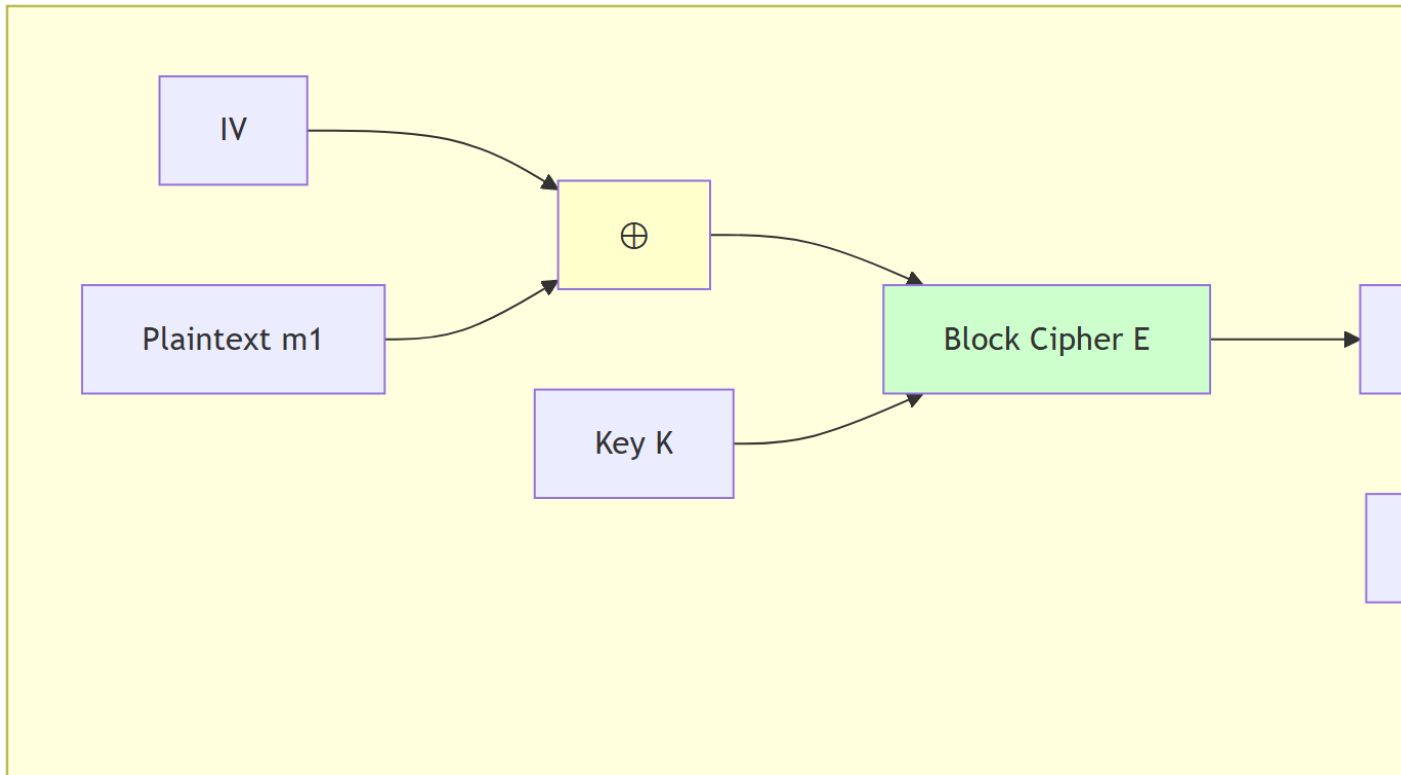
2.2 Cipher Block Chaining (CBC)

Principe : chaque bloc de plaintext est **XORé** avec le **ciphertext précédent** avant encryption.

$$c_i = E_K(m_i \oplus c_{i-1})$$

$$m_i = D_K(c_i) \oplus c_{i-1}$$

Avec $c_0 = IV$ (Initialization Vector)



Caractéristiques :

- **Plaintexts identiques** → ciphertexts différents (si IV change)
- **Patterns effacés** : chaînage masque la structure
- **Propagation d'erreurs limitée** : erreur sur c_j affecte m_j et m_{j+1} uniquement
- **Non parallélisable** en encryption (séquentiel)
- **Parallélisable** en décryption

IV (Initialization Vector) :

- Doit être **aléatoire** ou **pseudo-aléatoire**
- Peut être transmis **en clair**
- Doit être **différent** pour chaque message avec la même clé

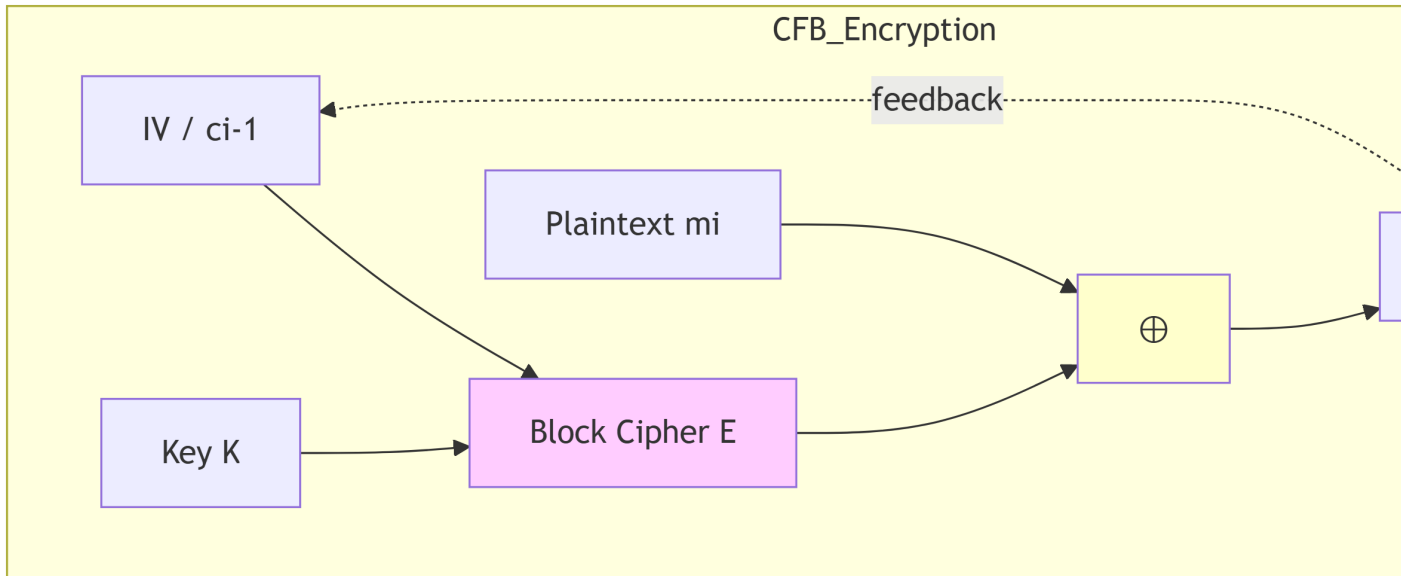
2.3 Cipher Feedback Mode (CFB)

Principe : fonctionne comme un **stream cipher** où le keystream est généré par le block cipher. Le keystream dépend des **ciphertexts précédents** (mode **asynchrone**).

$$c_i = m_i \oplus E_K(c_{i-1})$$

$$m_i = c_i \oplus E_K(c_{i-1})$$

Avec $c_0 = IV$



Caractéristiques :

- **Plaintexts identiques** → ciphertexts différents (si IV change)
- **Chaînage** : dépendances entre ciphertexts
- **Propagation d'erreurs** : erreur sur c_j affecte $\frac{n}{r}$ blocs suivants
 - n = taille nominale du block cipher
 - r = taille des plaintexts
- **Non parallélisable**
- **IV non confidentiel** mais doit être transmis

Usage : adapté aux transmissions avec pertes de paquets fréquentes

2.4 Output Feedback Mode (OFB)

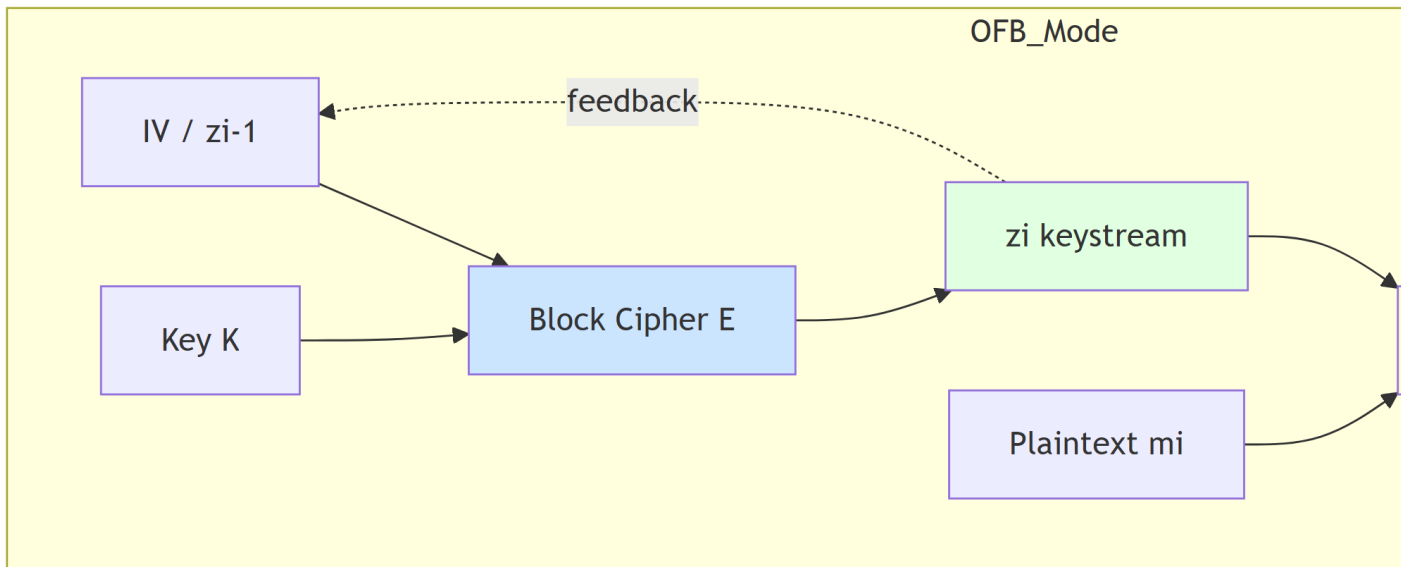
Principe : fonctionne comme un **stream cipher synchrone**. Le keystream est **entièrement déterminé** par la clé et l'IV, **indépendant** du plaintext et du ciphertext.

$$z_i = E_K(z_{i-1})$$

$$c_i = m_i \oplus z_i$$

$$m_i = c_i \oplus z_i$$


Avec $z_0 = IV$



Caractéristiques :

- **Plaintexts identiques** → ciphertexts différents (si IV change)
- **Pas de propagation d'erreurs** : erreur sur c_j n'affecte que m_j
- **Keystream pré-calculable** : efficace
- **CRITIQUE** : ne JAMAIS réutiliser le même IV avec la même clé (sinon keystream identique)
- **Parallélisable** si keystream pré-calculé

Attention réutilisation : Modifier l'IV pour chaque nouveau message !

 Texte original (Modes CFB et OFB)

Modes CFB et OFB: Caractéristiques


Les modes **CFB** et **OFB** fonctionnent comme un **stream cipher** avec un **keystream** généré par le **bloc de cryptage**. Dans **CFB**, le keystream dépend des **ciphertexts précédents** (**asynchrone**) alors que dans **OFB**, le keystream est **entièrement déterminé** par la clé et le **IV** (**synchrone**).

Particularités de CFB:

- Comme dans le mode CBC, des **plaintext identiques** sont traduits en **ciphertexts différents** si le **IV change**. Le **IV n'est pas nécessairement confidentiel** et peut être échangé en clair entre les parties.
- Le **chaînage** introduit également des **dépendances** entre les ciphertexts courants et les ciphertexts précédents. En particulier, si n est la **taille nominale de l'algorithme** et r est la **taille des plaintexts**, le ciphertext courant dépendra des $\frac{n}{r}$ **ciphertexts précédents** (chaque itération décalera l'entrée fautive de r positions, après $\frac{n}{r}$ itérations le ciphertext fautif sera "expulsé" complètement).
- La **propagation d'erreurs** obéit au même principe: une erreur dans un ciphertext se traduira par une mauvaise decryption des $\frac{n}{r}$ ciphertexts suivants.

Particularités de OFB:

- OFB a un comportement **identique** aux modes CBC et CFB pour l'**encryption de plaintext identiques**.
- **Pas de propagation d'erreurs** sur les ciphertexts adjacents.
- **Modifiez le IV** si la clé ne change pas pour **éviter la réutilisation du keystream** !!!

 Révision rapide (Modes CFB/OFB)

CFB (asynchrone) : keystream = $f(\text{ciphertexts précédents})$. Propagation erreur limitée ($\frac{n}{r}$ blocs).

OFB (synchrone) : keystream = $f(\text{clé} + \text{IV uniquement})$. Pas propagation erreur.

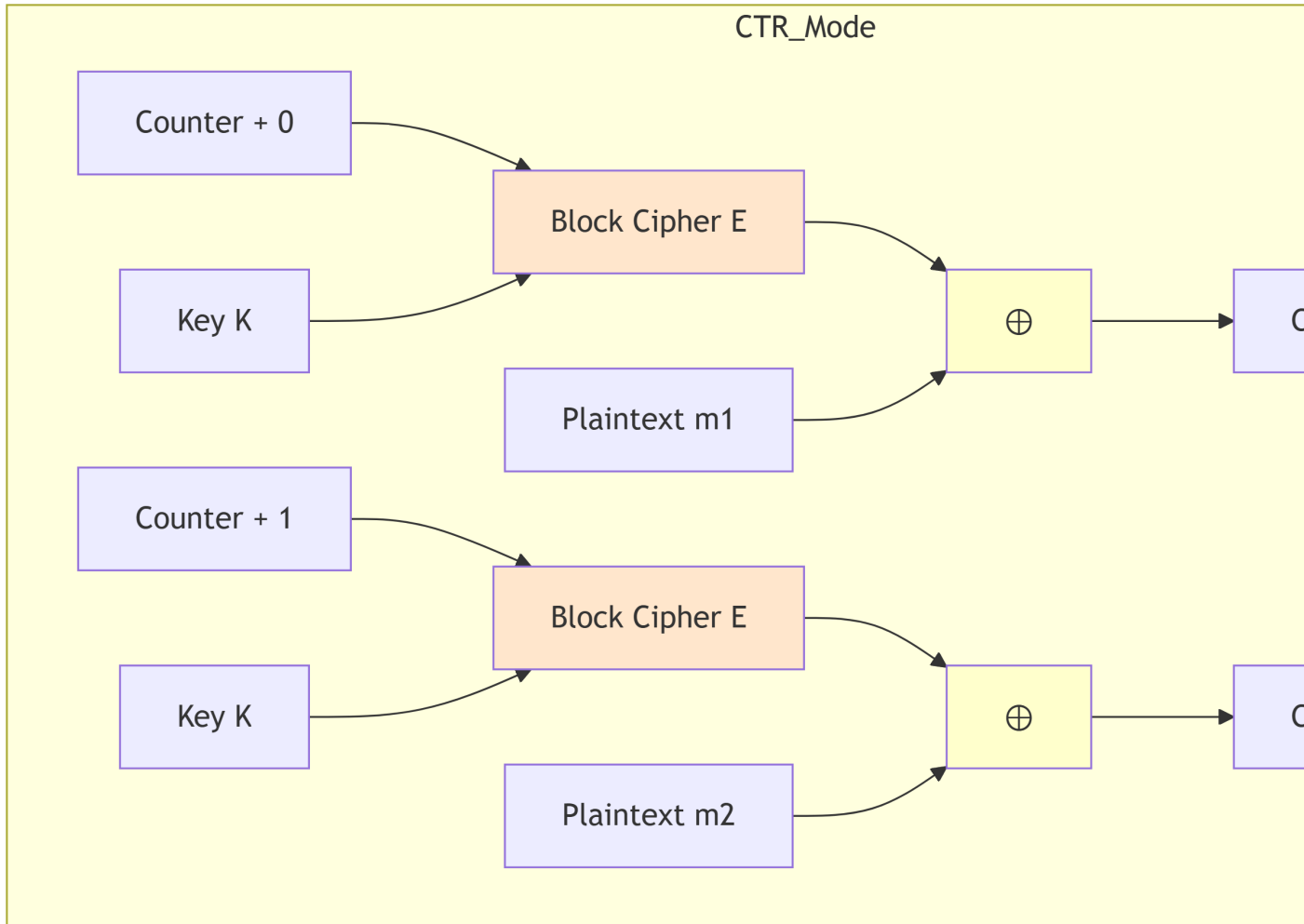
CRITIQUE : ne JAMAIS réutiliser même IV avec même clé. IV transmissible en clair.

2.5 Counter Mode (CTR)

Principe : le keystream est généré par l'**encryption d'un compteur** incrémenté à chaque bloc.

$$c_i = m_i \oplus E_K(\text{counter} + i)$$

$$m_i = c_i \oplus E_K(\text{counter} + i)$$



Caractéristiques :

- **Mode synchrone** : keystream = $f(\text{compteur})$
- **Parallélisable** : keystream pré-calculable pour encryption ET decryption
- **Accès aléatoire** : chaque bloc décryptable indépendamment
- **Pas de propagation d'erreurs**
- **Profite des architectures SIMD** : pas de dépendances entre blocs
- **Compteur** : doit être de taille 2^b (b = taille du bloc)
- **CRITIQUE** : ne JAMAIS réutiliser le même compteur avec la même clé

Gestion du compteur :

- **Incrémenter modulo 2^b** après chaque itération
- **Solution** : toujours incrémenter pour chaque flux encrypté
- Premier bloc du flux $i + 1 >$ dernier bloc du flux i

Applications :

- **ATM** (Asynchronous Transfer Mode)
- **IPsec** (IP security)
- **Lignes à haut débit** : transmission sélective des blocs
- **Transferts de grands volumes** : vidéo

Texte original (Counter Mode)

Counter Mode (CTR Mode)

Fréquemment utilisé comme support d'encryption dans des protocoles de transfert de données comme **ATM** (Asynchronous Transfer Mode) et **IPsec** (IP security).

Counter Mode (II)

- Le **keystream** est généré par l'**encryption d'un compteur aléatoire** de taille 2^b (avec b la taille du bloc) et nécessaire pour la décryption. Ce compteur est **incrémenté modulo 2^b** après chaque itération.
- Travaille en **mode synchrone**. La **réutilisation d'un même compteur** se traduit par un **keystream identique** !
- **Solution**: Toujours **incrémenter le compteur** pour chaque flot encrypté de telle sorte que le compteur du premier bloc d'un flot soit **plus grand que le dernier bloc** du flot précédent.
- **Facilement parallélisable**: Le keystream peut être **pré-calculé** aussi bien pour l'encryption que pour la décryption. Profite pleinement des **architectures SIMD** car contrairement aux autres modes de chaînage il n'y a pas des **dépendances entre les opérations** des différents blocs.
- **Accès aléatoire** à l'encryption/décryption de chaque bloc: Contrairement aux autres modes de chaînage où la i -ème opération dépend de la $(i - 1)$ -ème opération.
- Si à ceci on ajoute l'**absence de propagation d'erreurs**, le mode compteur facilite la (re)**transmission sélective** des blocs de ciphertext, ce qui le rend très attractif pour la **sécurisation de lignes à haut débit** ainsi que pour les **transferts encryptés de grands volumes** d'information (p.ex. vidéo).

Révision rapide (Counter Mode)

CTR : keystream = $E_K(\text{compteur} + i)$.

Avantages : parallélisable (encryption + décryption), accès aléatoire, pas propagation erreur, SIMD-friendly.

CRITIQUE : ne jamais réutiliser compteur.
Usage : ATM, IPsec, haut débit, vidéo.

3. Product Ciphers et Feistel Ciphers

Product Ciphers

Définition : schéma de cryptage combinant une **série de transformations successives** pour renforcer la résistance à la cryptanalyse.

Transformations courantes :

- Transpositions (permutations)
- Substitutions (S-boxes)
- XORs
- Combinaisons linéaires
- Multiplications modulaires

Feistel Ciphers

Définition : product cipher itératif avec structure spécifique.

Principe de fonctionnement :

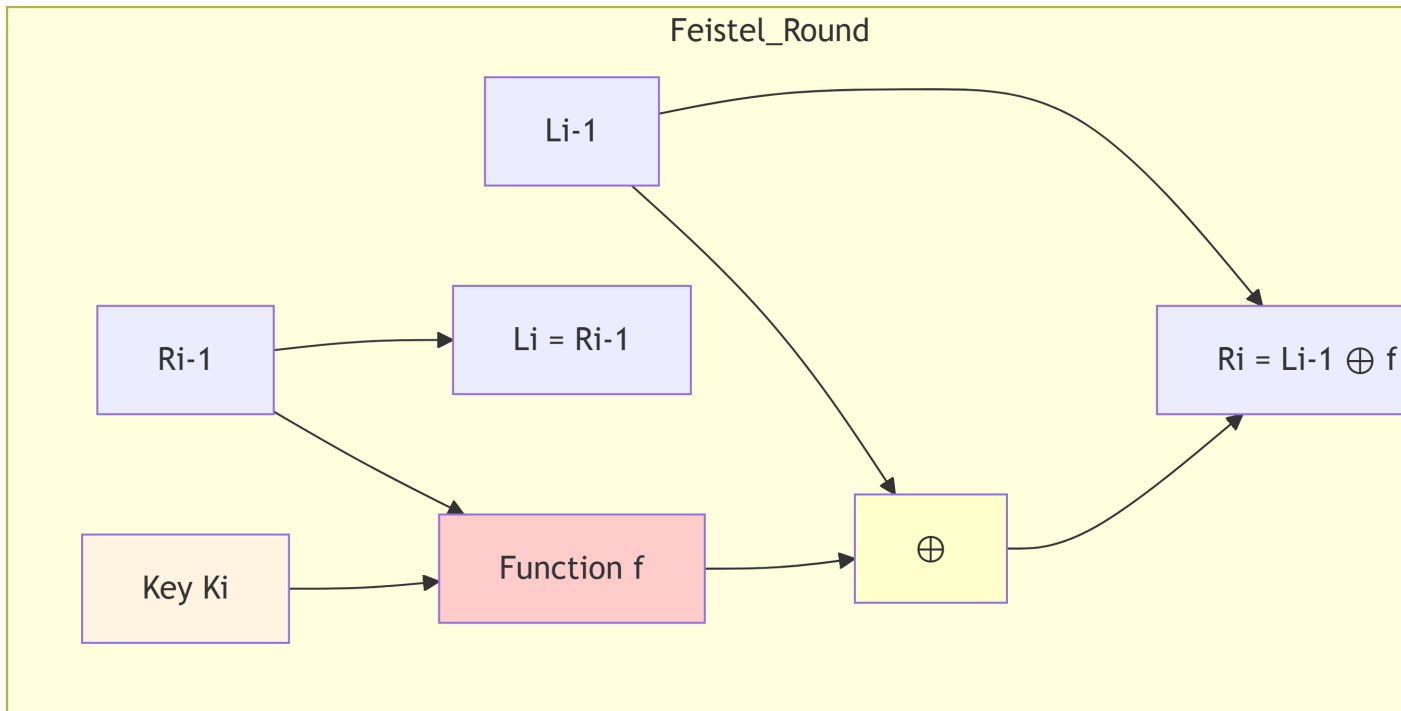
- **Entrée** : plaintext de $2t$ bits = (L_0, R_0) (deux sous-blocs de t bits)
- **Sortie** : ciphertext de $2t$ bits = (R_r, L_r) après r étapes (rounds)
- **Chaque étape** : bijection inversible (pour décryption unique)

Équations d'une étape i ($1 \leq i \leq r$) :

$$(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$$

Avec :

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



Caractéristiques :

- K_i : sous-clés générées à partir de la clé principale K
- Nombre d'étapes r : généralement **pair** et ≥ 3
 - Exemple : DES a 16 étapes
- **Permutation finale** : $(L_r, R_r) \rightarrow (R_r, L_r)$
- **Décryption** : identique à l'encryption mais sous-clés appliquées en **ordre inverse** (de K_r à K_1)

Opérations fréquentes :

- Permutations
- Substitutions (S-boxes)

i Texte original

Product Ciphers et Feistel Ciphers

- Un **product cipher** est un **schéma de cryptage** combinant une **série de transformations successives** dans le but de **renforcer la résistance à la cryptanalyse**. Des transformations courantes pour un product cipher sont: des **transpositions**, des **substitutions**, des **XORs**, des **combinaisons linéaires**, des **multiplications**

modulaires, etc.

- Un **Feistel cipher** est un **product cipher itératif** capable de transformer un **plaintext** de $2t$ bits de la forme (L_0, R_0) composé par deux **sous-blocs** L_0 et R_0 de t bits en un **ciphertext** de **taille** $2t$ de la forme (R_r, L_r) après r **étapes (rounds)** **successives** avec $r \geq 1$. **Chaque étape** définit une **bijection (inversible !)** pour permettre une decryption unique.
- Des **permutations** et des **substitutions** sont les opérations les plus fréquentes.
- Les étapes $1 \leq i \leq r$ s'écrivent: $(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$ avec $L_i = R_{i-1}$ et $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$. Les K_i sont des **sous-clés, différentes pour chaque étape**, générées à partir de la **clé principale** K du schéma de cryptage.
- Le **nombre d'étapes** propres à un Feistel cipher est normalement **pair** et ≥ 3 (p.ex. **DES a 16 étapes**)
- Après l'exécution de toutes les étapes, un Feistel cipher effectue une **permutation** des deux parties (L_r, R_r) en (R_r, L_r) .
- La **decryption** d'un Feistel Cipher est **identique à l'encryption** sauf que les sous-clés K_i sont appliquées en **ordre inverse** (De K_r à K_1).

Révision rapide

Product cipher : combinaison de transformations successives (transpositions, substitutions, XOR).

Feistel cipher :

- product cipher itératif
- plaintext $2t$ bits = (L_0, R_0)
- r rounds avec $L_i = R_{i-1}$ et $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$.
- Décryption = encryption avec sous-clés inversées.
- Exemple : DES (16 rounds).

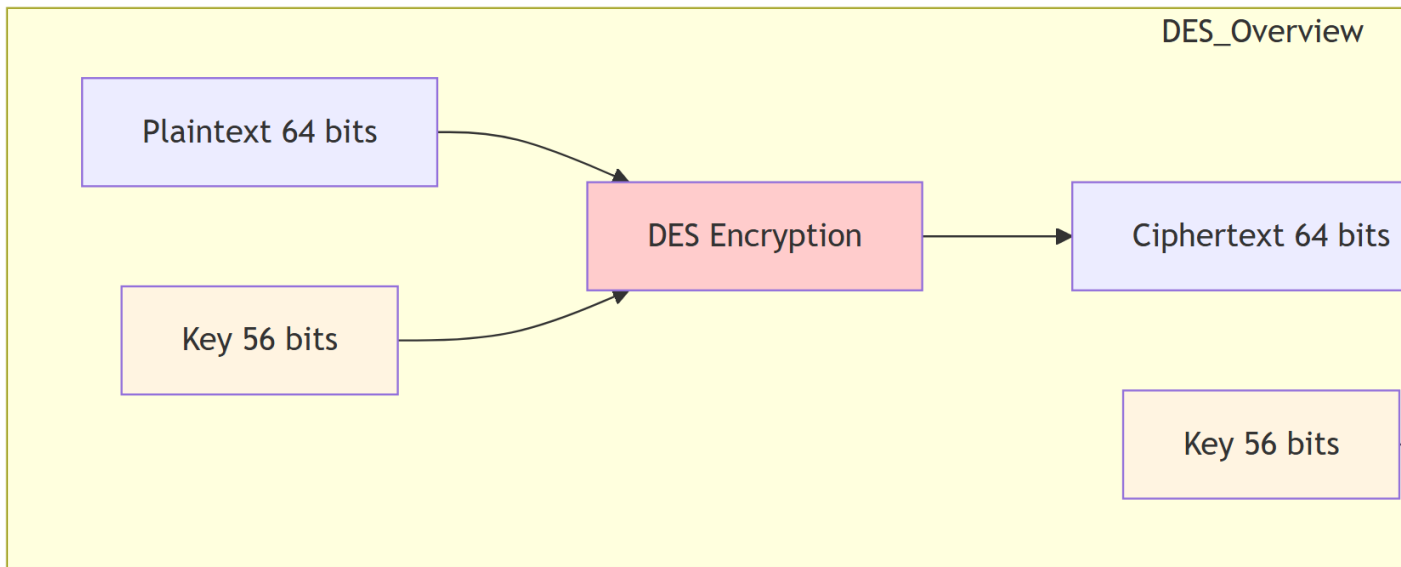
4. Data Encryption Standard (DES)

Présentation Générale

DES (Data Encryption Standard) : algorithme cryptographique le plus important jusqu'à l'avènement d'AES en 2001.

Caractéristiques principales :

- **Type** : Feistel Cipher
- **Taille des blocs** : 64 bits (taille nominale)
- **Taille de la clé** : 56 bits effectifs (64 bits totaux avec 8 bits de parité)
- **Nombre d'étapes** : 16 rounds
- **Sous-clés** : 16 sous-clés de 48 bits (une par étape)
- **Modes d'utilisation** : ECB, CBC, CFB, OFB



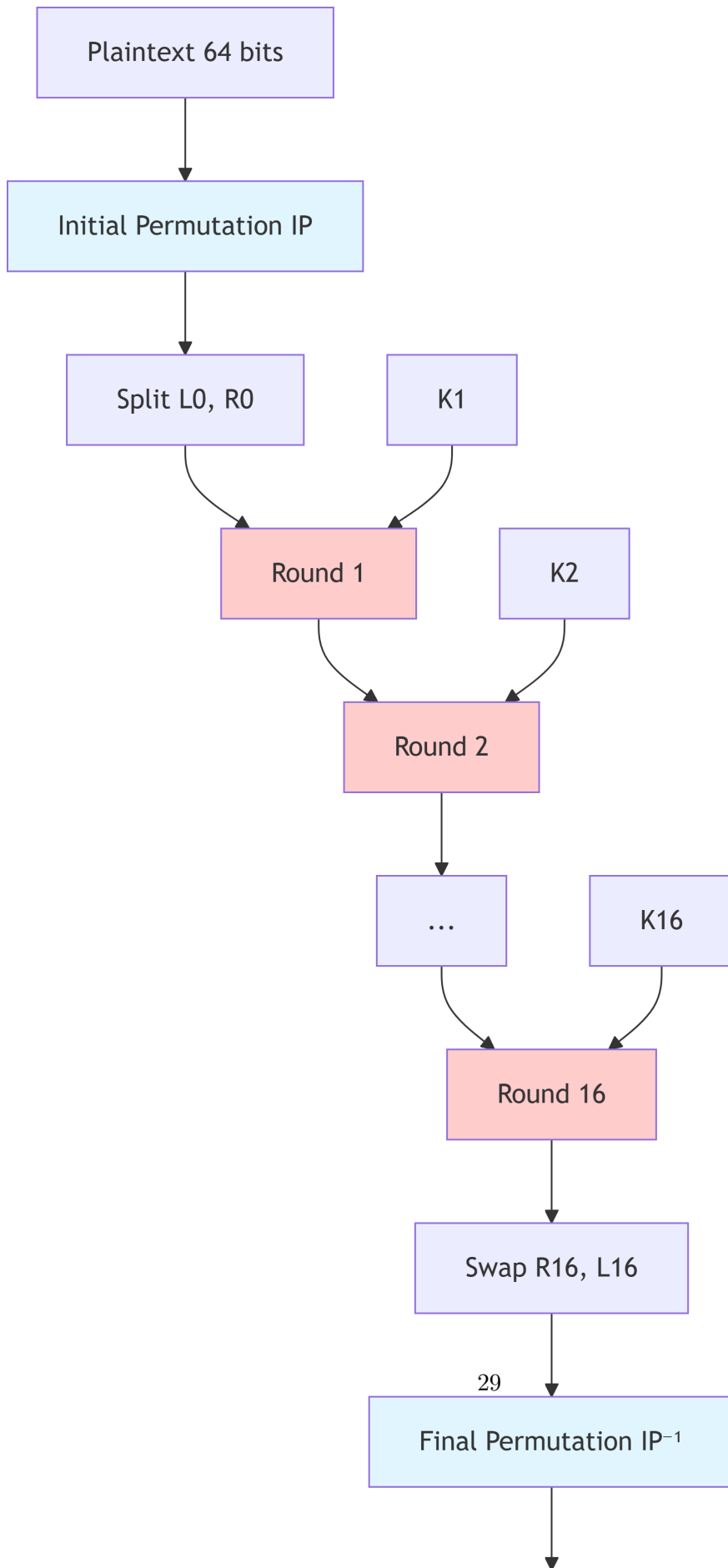
Structure de DES

Composants principaux :

1. **Permutation initiale (IP)** : permutation des 64 bits d'entrée
2. **16 rounds Feistel** : transformation itérative
3. **Permutation finale (IP⁻¹)** : inverse de IP

Chaque round applique :

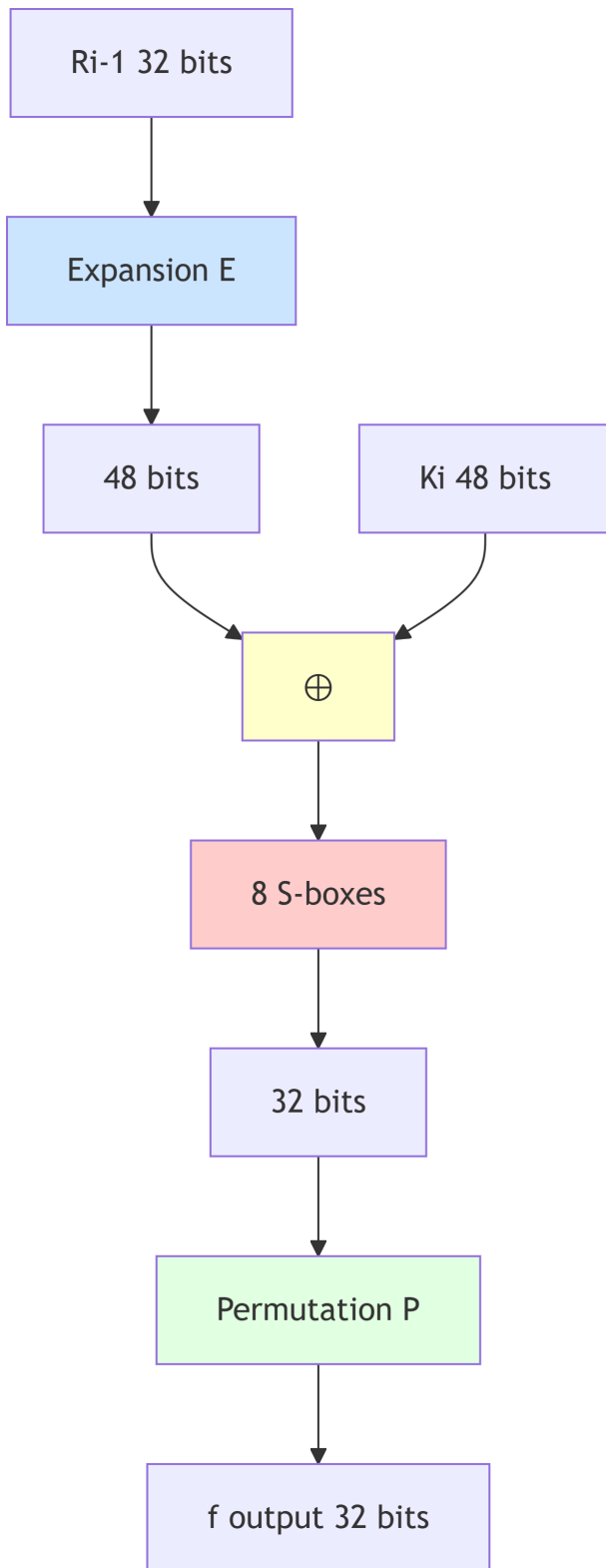
- Division en deux moitiés : L_{i-1} et R_{i-1} (32 bits chacune)
- Fonction f sur R_{i-1} avec sous-clé K_i
- XOR avec L_{i-1}
- Échange des moitiés



Fonction Cipher de DES

La fonction f pour chaque round :

1. **Expansion E** : 32 bits \rightarrow 48 bits (table E)
2. **Key Addition** : XOR avec sous-clé K_i (48 bits)
3. **S-boxes** : 8 S-boxes transforment 48 bits \rightarrow 32 bits
 - Chaque S-box : 6 bits entrée \rightarrow 4 bits sortie
4. **Permutation P** : permutation des 32 bits résultants



Fonctionnement des S-boxes :

Entrée : $a_1a_2a_3a_4a_5a_6$ (6 bits)

- **Ligne** : $a_1 + 2a_6$ (bits externes)
- **Colonne** : $a_2 + 2a_3 + 4a_4 + 8a_5$ (bits internes)
- **Sortie** : valeur de la cellule correspondante (4 bits)

Génération des Sous-clés

Processus :

1. Clé principale : 64 bits (56 effectifs + 8 parité)
2. **Permuted Choice 1 (PC-1)** : sélection de 56 bits
3. Division en deux moitiés : C_0 et D_0 (28 bits chacune)
4. Pour chaque round i :
 - Rotation circulaire gauche de C_{i-1} et D_{i-1}
 - **Permuted Choice 2 (PC-2)** : sélection de 48 bits pour K_i

Rotations :

- Rounds 1, 2, 9, 16 : 1 position
- Autres rounds : 2 positions

i Texte original (DES Fonctionnement)

DES: Fonctionnement

Cipher Fonction

- **Expansion E**: Les **32 bits de l'entrée** sont transformés en un vecteur de **48 bits** en utilisant la **table E**. La première ligne de cette table indique comment sera généré le premier sous-bloc de 6 bits: on prendra en premier le 32e bit et après les bits 1,2,3,4,5. Le deuxième sous-bloc commence par le 4e bit ensuite les bits 5,6,7,8,9 et ainsi de suite...
- **Key addition: XOR du vecteur de 48 bits** avec la clé.
- **S-boxes**: On applique **8 S-boxes** sur le vecteur de 48 bits résultant du XOR précédent. Chacune de ces S-boxes prend un **sous-bloc de 6 bits** et le transforme en un **sous-bloc de 4 bits**. L'opération s'effectue de la manière suivante: Si on dénote les 6 bits d'input de la S-box comme: $a_1a_2a_3a_4a_5a_6$. La sortie est donnée par le contenu de la cellule située dans la **ligne** $a_1 + 2a_6$ et la **colonne** $a_2 + 2a_3 + 4a_4 + 8a_5$.
- **Permutation P**: La permutation P fonctionne comme suit: Le premier bit est envoyé à la 16e position, le deuxième à la 7e position et ainsi de suite.

Permutations IP et IP⁻¹

- Agissent respectivement au **début** et à la **fin** du traitement du bloc et sur l'**ensemble des 64 bits**.

Révision rapide (DES)

DES : Feistel cipher, 64 bits blocs, 56 bits clé effective, 16 rounds.

Fonction f : Expansion E (32→48 bits) → XOR K_i → 8 S-boxes (48→32 bits) → Permutation P.

S-box : 6 bits input → 4 bits output via table (ligne = bits externes, colonne = bits internes).

Permutations : IP (initiale) et IP⁻¹ (finale) sur 64 bits.

5. Triple-DES et Sécurité de DES

Vulnérabilités de DES

Problème principal : taille de l'espace de clés $\{0,1\}^{56}$ insuffisante.

Attaque brute force :

- **1999** : clé trouvée en **24 heures**
- Technique : brute force massivement parallèle (100'000 PCs sur Internet)
- Known plaintext attack

Triple-DES (3DES)

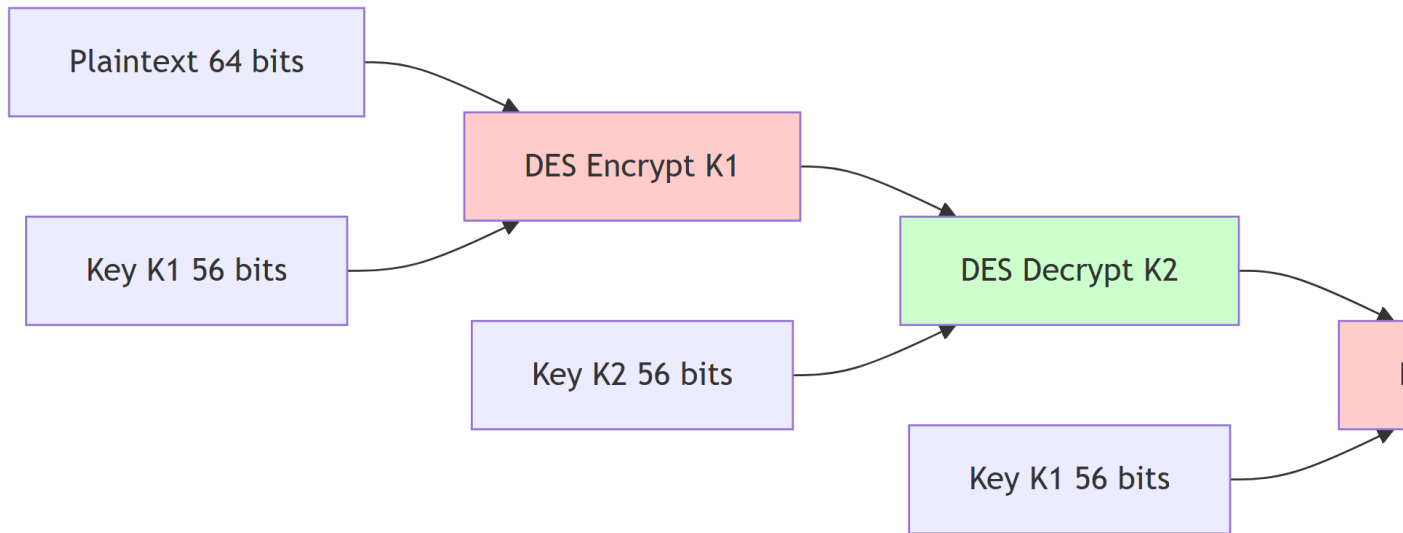
Solution : augmenter l'espace des clés à $\{0,1\}^{112}$.

Schéma :

$$C = E_{K_1}(D_{K_2}(E_{K_1}(P)))$$

Avec :

- E : encryption DES
- D : décryption DES
- K_1, K_2 : deux clés de 56 bits



Avantages :

- **Sécurité satisfaisante** : espace de clés 2^{112}
- **Compatibilité** : réutilisation du hardware/software DES existant
- **Migration progressive** : en attendant AES

Inconvénient :

- **Performances** : $3\times$ plus lent (3 exécutions DES successives)

Propriétés de DES

1. DES n'est pas un groupe

DES n'est PAS un groupe pour la composition :

$$\nexists K_3 \text{ tel que } E_{K_3}(E_{K_2}(E_{K_1}(x))) = E_{K_3}(x)$$

Conséquence : encryption composée (Triple-DES) augmente considérablement la sécurité.

Si DES était un groupe : recherche exhaustive sur $\{0, 1\}^{56}$ casserait l'algorithme indépendamment du nombre d'exécutions consécutives.

2. Clés faibles et semi-faibles

- **Clé faible** : $E_K(E_K(x)) = x$
- **Paire de clés semi-faibles** : $E_{K_1}(E_{K_2}(x)) = x$

Caractéristique : clés faibles génèrent des sous-clés identiques par paires :

- $k_1 = k_{16}, k_2 = k_{15}, \dots, k_8 = k_9$
- Facilite la cryptanalyse

DES a 4 clés faibles :

Clé faible (hexadécimal)
0101 0101 0101 0101
0101 0101 FEFE FEFE
FEFE FEFE FEFE FEFE
FEFE FEFE 0101 0101

Et 6 paires de clés semi-faibles

i Texte original (DES et 3DES)

DES et Triple-DES

- La **taille de l'ensemble de clés** ($\{0, 1\}^{56}$) constitue la **plus grande menace** qui pèse sur DES avec les ressources de calcul actuels. En **1999** il a suffi de **24 heures** pour trouver la clé à partir d'un **known plaintext** en utilisant une technique **brute force massivement parallèle** (100'000 PCs connectés sur Internet).
- **Triple DES** nous met à l'abri de ces **attaques brute force** en augmentant l'**espace des clés possibles** à $\{0, 1\}^{112}$.
- Cette alternative permet de continuer à utiliser les **“boîtes” DES** (hardware et software) en attendant une migration vers AES.
- Le **niveau de sécurité** obtenu par cette solution est **très satisfaisant**.
- L'**impact en termes de performances** de trois exécutions successives de DES reste un **inconvenient** pour certaines applications.

DES: propriétés

- **DES n'est pas un groupe** (au sens algébrique) avec la composition: En d'autres termes, DES étant une permutation: $\{0, 1\}^{64} \rightarrow \{0, 1\}^{64}$, si DES était un groupe pour la composition, ceci voudrait dire que: $\exists K_3$ t.q. $E_{K_3}(E_{K_2}(x)) = E_{K_3}(x)$
Cette propriété permet d'assurer que l'**encryption composée** (comme Triple-DES) **augmente considérablement la sécurité** de DES. Si DES était un groupe, la recherche exhaustive sur l'ensemble de clés possibles ($\{0, 1\}^{56}$) permettrait de “casser” l'algorithme **indépendamment du nombre d'exécutions consécutives** de DES.

- **Clés faibles et mi-faibles** (weak and semi-weak keys):
 - Une clé K est dite **faible** si $E_K(E_K(x)) = x$.
 - Une paire de clés (K_1, K_2) est dite **mi-faible** si $E_{K_1}(E_{K_2}(x)) = x$.
- Les clés faibles ont la particularité de générer de **sous-clés identiques par paires** ($k_1 = k_{16}, k_2 = k_{15}, \dots, k_8 = k_9$), ce qui **facilite la cryptanalyse**.
- **DES a 4 clés faibles** (et 6 paires de clés mi-faibles).

💡 Révision rapide (3DES et sécurité)

Vulnérabilité DES : espace clés 2^{56} cassable en 24h (1999). **Triple-DES** : $E_{K_1}(D_{K_2}(E_{K_1}(P)))$, espace 2^{112} , réutilise hardware DES, $3\times$ plus lent. **DES groupe** \rightarrow encryption composée renforce sécurité. **4 clés faibles** générant sous-clés identiques par paires \rightarrow facilite cryptanalyse.

6. Advanced Encryption Standard (AES)

Présentation Générale

AES (Advanced Encryption Standard) : standard adopté en novembre 2001.

Conception : Johan Daemen et Vincent Rijmen (nom original : **Rijndael**)

Caractéristiques principales :

- **Type** : block cipher itératif (mais **PAS un Feistel Cipher**)
- **Taille des blocs** : 128 bits
- **Taille de clé variable** : 128, 192 ou 256 bits
- **Nombre de rounds** : dépend de la taille de clé
 - 10 rounds pour clé 128 bits
 - 12 rounds pour clé 192 bits
 - 14 rounds pour clé 256 bits
- **Modes d'utilisation** : ECB, CBC, CFB, OFB, CTR

Avantages par rapport à DES :

- **Processus ouvert** : consultation et analyse par experts mondiaux
- **$\sim 2\times$ plus performant** en software
- **$\sim 10^{22}$ fois plus sûr** (théoriquement)

- **Évolutif** : taille de clé augmentable si nécessaire

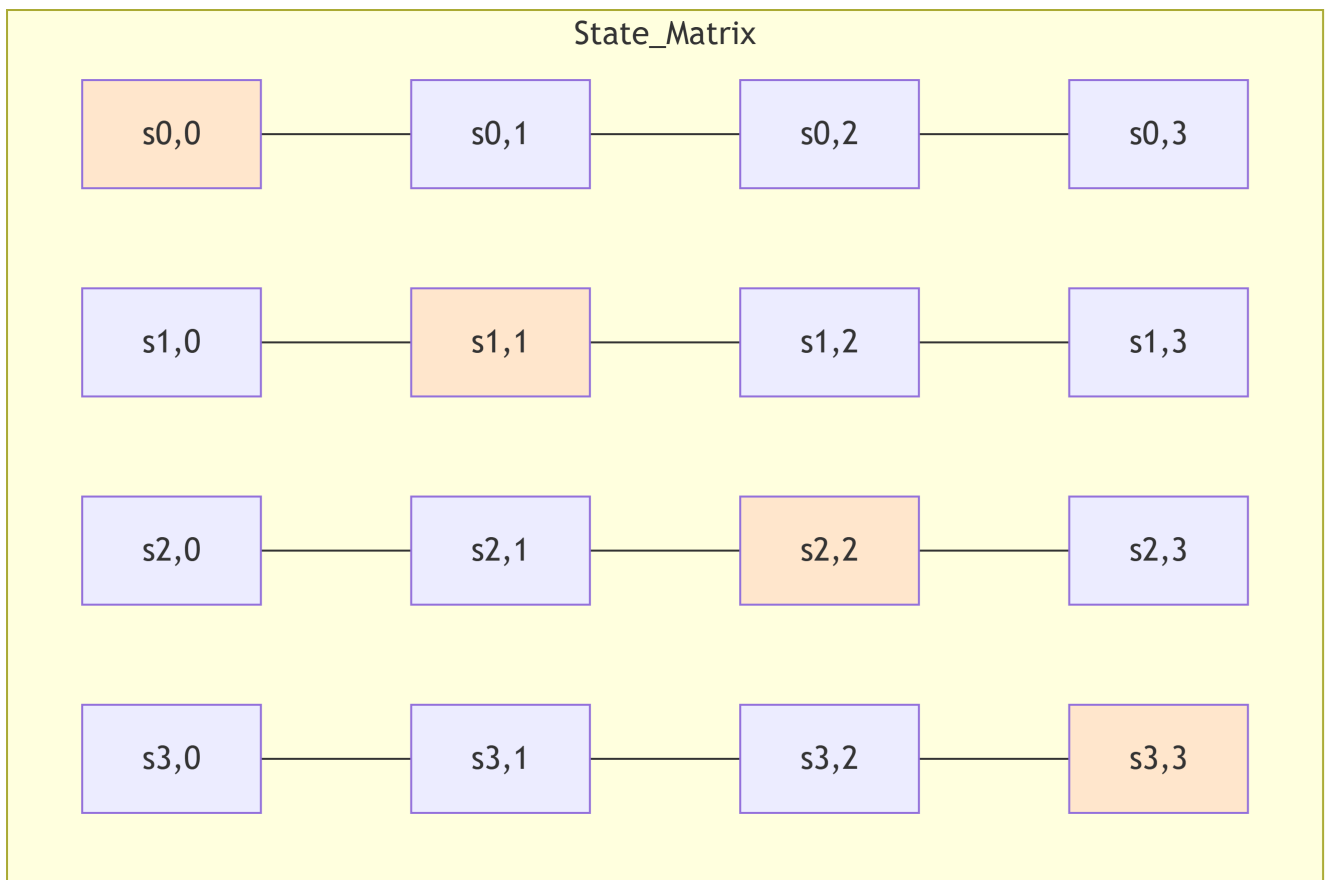
Structure d'AES

Unité de base : matrice **State** de 4 lignes \times 4 colonnes (pour clé 128 bits)

- Chaque élément = 1 byte
- **Total** : 16 bytes = 128 bits

Opérations sur le corps $GF(2^8)$:

- Byte = élément de $GF(2^8)$
- Corps fini de polynômes de degré 7 avec coefficients dans $GF(2)$
- Additions, multiplications définies dans $GF(2^8)$



Détail d'un Round AES

Quatre opérations par round :

1. SubBytes (ByteSub)

- Substitution non linéaire via **S-box**
- Chaque byte transformé indépendamment
- Résistance à la cryptanalyse linéaire et différentielle

2. ShiftRows

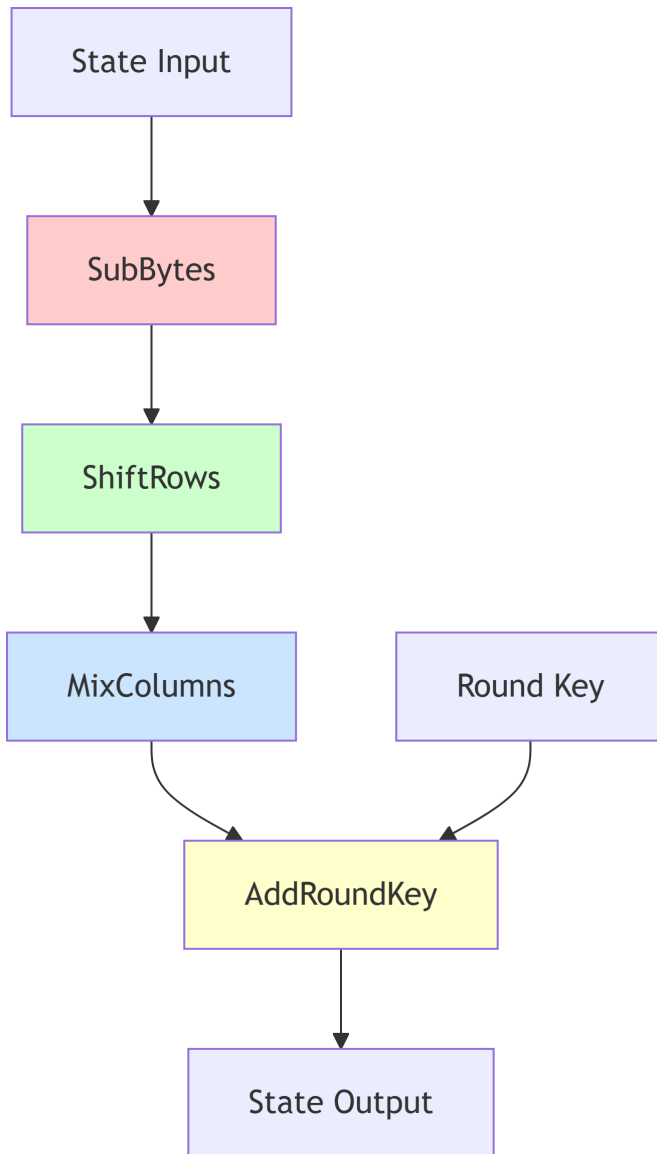
- **Permutation des bytes** avec décalages variables par ligne
- Ligne 0 : pas de décalage
- Ligne 1 : décalage gauche 1 position
- Ligne 2 : décalage gauche 2 positions
- Ligne 3 : décalage gauche 3 positions

3. MixColumns

- Chaque colonne = combinaison linéaire des autres colonnes
- **Multiplication de matrices** dans $GF(2^8)$
- Diffusion maximale

4. AddRoundKey

- **XOR** de la matrice State avec la sous-clé du round
- Sous-clé = résultat du Key Schedule



Round final : identique SAUF pas de MixColumns

Key Schedule (Génération des Sous-clés)

Processus :

1. **Key Expansion** : génération d'une matrice étendue
 - Clé 128 bits \rightarrow matrice $4 \times 4 \times (N_e + 1)$ bytes
 - N_e = nombre de rounds

2. Key Selection : extraction des sous-clés

- Première sous-clé : 4 premières colonnes
- Deuxième sous-clé : 4 colonnes suivantes
- Etc.

Opérations :

- Rotations de bytes
- Substitutions via S-box
- XOR avec constantes (Rcon)

Pseudo-code AES

```
Rijndael(State, CipherKey) {  
    KeyExpansion(CipherKey, ExpandedKey); // Key Schedule  
  
    AddRoundKey(State, ExpandedKey[0..3]); // XOR initial  
  
    for(i = 1; i < Ne; i++) {  
        Round(State, ExpandedKey[4*i...(4*i)+3]);  
    }  
  
    FinalRound(State, ExpandedKey[4*Ne...4*Ne+3]); // Sans MixColumns  
}
```

Décryption AES

Principe : appliquer les **opérations inverses** dans chaque round.

Opérations inverses :

- **InvSubBytes** : substitution inverse via S-box ¹
- **InvShiftRows** : décalages droite (au lieu de gauche)
- **InvMixColumns** : multiplication matricielle inverse
- **AddRoundKey** : auto-inverse (XOR)

Ordre : inverse de l'encryption avec sous-clés en ordre inverse

i Texte original (AES)

Advanced Encryption Standard (AES)

- Adopté comme **standard en Novembre 2001**, conçu par **Johan Daemen et Vincent Rijmen** (d'où son nom original **Rijndael**).

- Il s'agit également d'un **block cipher itératif** (comme DES) mais **pas d'un Feistel Cipher**.
- **Blocs Plaintext/Ciphertext: 128 bits.**
- **Clé de longueur variable: 128, 192, ou 256 bits.**
- Contrairement à DES, AES est issu d'un **processus de consultation et d'analyse ouvert** à des experts mondiaux.
- Techniques semblables à DES (substitutions, permutations, XOR...) complémentées par des **opérations algébriques simples** et très performantes.
- Toutes les opérations s'effectuent dans le **corps $GF(2^8)$** : le corps fini de **polynômes de degré 7** avec des **coefficients dans $GF(2)$** .
- En particulier, un **byte pour AES est un élément dans $GF(2^8)$** et les **opérations sur les bytes** (additions, multiplications,...) sont **définies comme sur $GF(2^8)$** .
- **~2 fois plus performant** (en software) et **~ 10^{22} fois (en théorie...) plus sûr** que DES...
- **Évolutif**: La taille de la clé peut être augmentée si nécessaire.

Détail d'une Etape (round) AES

L'unité de base sur laquelle s'appliquent les calculs est une **matrice de 4 lignes et 4 colonnes** (dans le cas d'une clé de 128 bits) dont les éléments sont des **bytes**:

- **ByteSub**: Opération **non linéaire (S-box)** conçu pour **résister à la cryptanalyse linéaire et différentielle**.
- **ShiftRow**: **Permutation des bytes** introduisant des **décalages variables** sur les lignes.
- **MixColumn**: Chaque colonne est remplacée par des **combinaisons linéaires** des autres colonnes (**multiplication des matrices !**)
- **AddRoundKey**: **XOR** de la matrice courante avec la **sous-clé** correspondante à l'étape courante.

AES: Fonctionnement Global

- Le **nombre d'étapes** d'AES varie en fonction de la **taille de la clé**. Pour une clé de **128 bits**, il faut effectuer **10 étapes**. Chaque augmentation de 32 bits sur la taille de la clé, entraîne une **étape supplémentaire** (14 étapes pour des clés de 256 bits).

- La **decryption** consiste en appliquer les **opérations inverses** dans chacune des étapes (**InvSubBytes**, **InvShiftRows**, **InvMixColumns**). **AddRoundKey** (à cause du XOR) est sa **propre inverse**.
- Le **Key Schedule** consiste en:
 - Une opération d'**expansion de la clé** principal. Si N_e est le nombre d'étapes (dépendant de la clé), une **matrice de 4 lignes et $4 \times (N_e + 1)$ colonnes** est générée.
 - Une opération de **sélection de la clé d'étape**: La **première sous-clé** sera constituée des **4 premières colonnes** de la matrice générée lors de l'expansion et ainsi de suite.



Révision rapide (AES)

AES (Rijndael 2001) : block cipher itératif (PAS Feistel), 128 bits blocs, clés 128/192/256 bits \rightarrow 10/12/14 rounds.

State : matrice 4×4 bytes dans $GF(2^8)$.

4 opérations/round :

- SubBytes (S-box non linéaire)
- ShiftRows (décalages lignes)
- MixColumns (combinaisons linéaires)
- AddRoundKey (XOR sous-clé).

2× plus rapide que DES, 10^{22} fois plus sûr.

7. Attaques et Sécurité d'AES

Forces d'AES

Simplicité et performances :

- Algorithme simple et efficace
- Fonctionne sur plateformes limitées (cartes à puce 8 bits)
- Optimisations hardware et software

Attaques Publiées

1. Attaques algébriques (2002)

Technique XSL (N. Courtois et P. Pieprzyk) :

- Représente AES comme **système de 8000 équations quadratiques** avec 1600 inconnues binaires
- **Effort estimé** : 2^{100} opérations (encore une conjecture)
- **Caractéristique** : nécessite peu de known plaintexts
- **Distinction** : différent des attaques linéaires/différentielles

Critique : basées sur le caractère “fortement algébrique” d’AES (largement contesté)

2. Related Key Attacks (2009-2011)

Principe : attaques basées sur des **clés similaires**

- Résultats intéressants sur **versions réduites** d’AES
- Ne compromettent pas AES complet

3. Side Channel Attacks

Principe : attaques sur l’**implémentation** (pas l’algorithme)

Techniques :

- **Cache timing attacks** : analyse des accès cache
- **Power analysis** : consommation électrique
- **Electromagnetic analysis** : émissions électromagnétiques

Exemple (2005) : Osvik, Shamir, Tromer

- Extraction de clé 128 bits avec **6-7 couples plaintext/ciphertext**
- Basée sur analyse des **accès cache**

4. Meet in the Middle sur structures bicycliques (2011-2015)

Résultat :

- Réduit l’effort pour AES-128 à 2^{126} (facteur 4 vs brute force)
- **Reste largement au-dessus** des capacités actuelles

Sécurité Pratique

Hypothèse fondamentale : clé d'entropie maximale

Attaques récentes (WPA2, etc.) :

- Exploitent la **faiblesse des passwords/passphrases**
- Pas de faille dans AES lui-même
- Problème : génération de clés depuis passwords faibles

Rappel critique : qualité de la clé = sécurité du système

i Texte original (Attaques AES)

AES: Remarques Finales et Attaques (I)

- La plus grande **force de AES** réside dans sa **simplicité** et dans ses **performances**, y compris sur des plate-formes à **capacité de calcul réduite** (p.ex. des **cartes à puces** avec des processeurs à 8 bits).
- Depuis sa publication officielle, des **nombreux travaux de cryptanalyse** ont été publiés avec des résultats très intéressants. En particulier, **N. Courtois et P.Pieprzyk** ont présenté une technique appelée **XSL** permettant de représenter AES comme un **système de 8000 équations quadratiques** avec **1600 inconnues binaires**. L'**effort nécessaire** pour casser ce système est estimé (il s'agit encore d'une **conjecture...**) à 2^{100} .
- Ces attaques se basent sur le **caractère fortement algébrique** (et largement contesté...) de AES. De plus, il suffit de **quelques known plaintexts** pour les mettre en place, ce qui les distingue des attaques linéaires et différentielles.
- Ces dernières années (2009-2011) des **attaques basées sur des clés similaires** (related key attacks) ont obtenu des résultats intéressants sur des **versions réduites** d'AES.
- Une autre famille d'attaques dénommée **side channel attacks** agissant directement sur l'**implémentation de l'algorithme** permet d'extraire des informations d'intérêt cryptographique lors de l'exécution de l'encryption.

AES: Remarques finales et Attaques (II)

- En **2015** une attaque de type **Meet in the Middle** basé sur des **structure bicycliques** a montré qu'il était possible de réduire l'**effort nécessaire** pour trouver une clé AES-128 à 2^{126} , soit un **facteur 4** par rapport au brute force. Ceci reste tout de même **largement au dessus** des capacités de calcul actuelles.

- Une autre famille d'attaques dénommée **side channel attacks** agissant directement sur l'**implémentation de l'algorithme** permet d'extraire des informations d'intérêt cryptographique lors de l'exécution de l'encryption. En particulier, les auteurs arrivent à **extraire la clé de 128 bits** avec seulement **6-7 couples plaintext/ciphertexts** en se basant sur les **accès cache**.
- La **sécurité de AES** (comme pour tout autre algorithme d'encryption) se base toujours sur l'hypothèse d'une **clé d'entropie maximale**. Les **attaques publiées récemment** sur des protocoles basés sur AES (comme WPA2) exploitent la **faiblesse des passwords/passphrases** qui sont à l'origine des clés utilisées.

Révision rapide (Sécurité AES)

Forces : simplicité, performances (même cartes 8 bits). **Attaques** : XSL (2^{100} , algébrique), related keys (versions réduites), side channel (implémentation, cache), Meet-in-Middle bicyclic (2^{126}). **Sécurité** : hypothèse clé entropie max. Attaques pratiques = passwords faibles, pas faille AES.

8. Techniques de Cryptanalyse des Block Ciphers

8.1 Cryptanalyse Différentielle

Principe : attaque **chosen plaintext** analysant la **propagation des différences** entre deux plaintexts à travers les rounds.

Méthode :

1. Choisir deux plaintexts avec différence connue : x_a et x_b
2. Observer la propagation : $\Delta x = x_a \oplus x_b$
3. Analyser les ciphertexts : $\Delta y = y_a \oplus y_b$
4. **Attribuer des probabilités aux clés** selon les changements observés
5. **Clé la plus probable** = clé correcte (après nombreux essais)

Caractéristiques :

- Nécessite 2^{47} **couples chosen plaintext** pour DES
- **Probabilités** : dépendent des S-boxes et de la structure
- Plus le nombre de couples augmente, plus la probabilité de succès augmente

Sensibilité : très sensible au **nombre de rounds**

- Chances de réussite augmentent **exponentiellement** quand rounds diminuent

8.2 Cryptanalyse Linéaire

Principe : attaque **known plaintext** créant un **simulateur linéaire** du block cipher.

Méthode :

1. Créer des **approximations linéaires** de l'algorithme
2. Analyser un grand nombre de paires plaintext/ciphertext
3. Les bits de la clé du simulateur **tendent à coïncider** avec ceux de la clé réelle (calcul probabiliste)

Complexité pour DES :

- 2^{38} **known plaintexts** \rightarrow probabilité 10% de deviner juste
- 2^{43} **known plaintexts** \rightarrow probabilité 85% de succès

Caractéristiques :

- **Attaque analytique la plus puissante** à ce jour sur block ciphers
- Aussi **sensible au nombre de rounds**

8.3 Comparaison Différentielle vs Linéaire

Difficultés communes :

- **Parallélisation** : moins efficace que brute force parallèle
- **Sensibilité aux rounds** : efficacité diminue exponentiellement avec le nombre de rounds

DES et ces attaques :

- Conjecture répandue : concepteurs de DES **connaissaient ces attaques** (années 1970, inédites à l'époque)
- **Design des S-boxes** : résistance très grande aux deux techniques

8.4 Attaque Meet-in-the-Middle

Principe : exploite les constructions **composées** du type $y = E_{K_2}(E_{K_1}(x))$.

Méthode :

1. Construire liste L_1 : $L_1 = \{E_{K_1}(x) \mid K_1 \in \text{KeySpace}\}$
2. Construire liste L_2 : $L_2 = \{D_{K_2}(y) \mid K_2 \in \text{KeySpace}\}$
3. Identifier **éléments répétés** dans L_1 et L_2
4. Vérifier hypothèse avec **deuxième known plaintext**
5. Les clés K_1 et K_2 associées sont probablement les clés recherchées

Exemple pour DES :

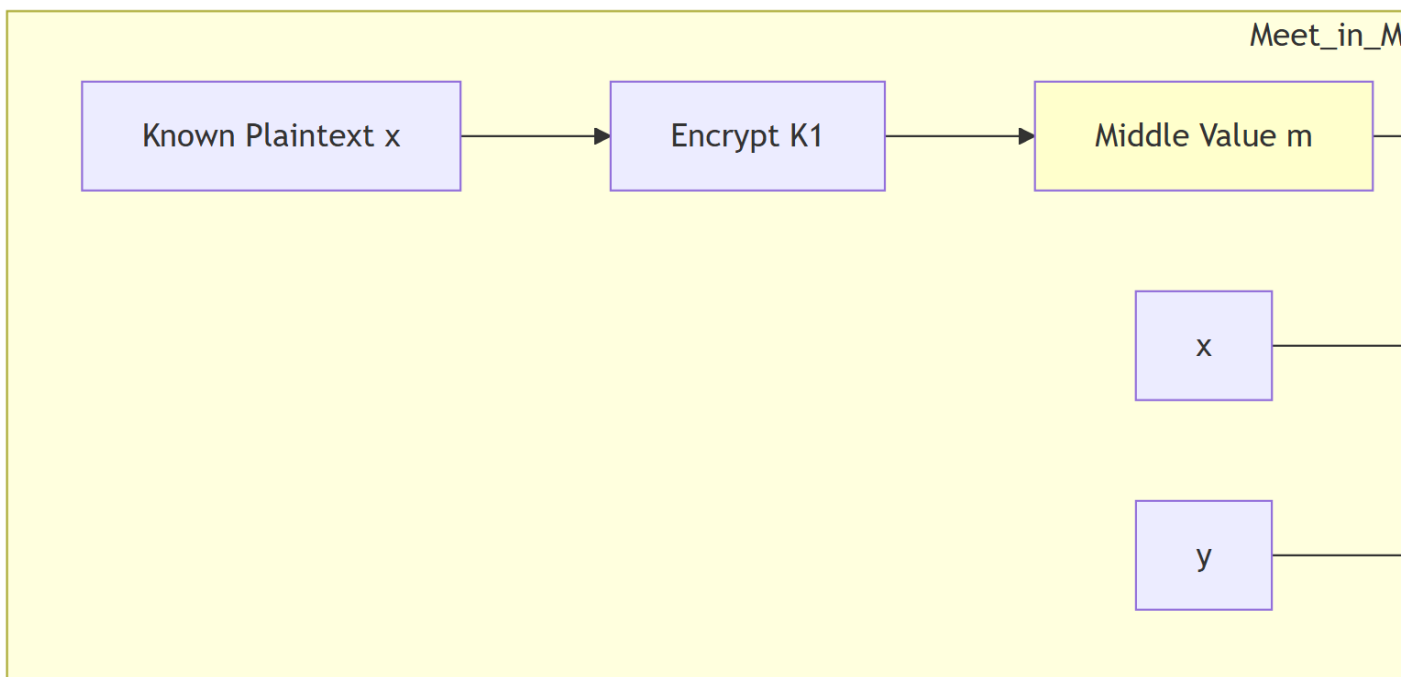
Espace de clés intuitif pour $E_{K_2}(E_{K_1}(x)) : \{0, 1\}^{112}$

Effort réel :

- 2^{57} **opérations** pour établir les deux listes
- 2^{56} **blocs** de 64 bits de stockage
- **Nettement inférieur** au 2^{112} estimé intuitivement

Applications :

- Attaques sur **constructions composées**
- Cryptanalyse **interne** des block ciphers



i Texte original (Cryptanalyse)

Techniques de Cryptanalyse sur les Block Ciphers

Cryptanalyse Différentielle

- Il s'agit d'une **attaque chosen plaintext** qui s'intéresse à la **propagation des différences** dans deux plaintexts au fur et à mesure qu'ils évoluent dans les différentes étapes de l'algorithme.
- Il **attribue des probabilités aux clés** qu'il "devine" en fonction des **changements**

qu'elles induisent sur les ciphertexts. La **clé la plus probable** a des bonnes chances d'être la clé correcte après un **grand nombre** de couples plaintext/ciphertext.

- Nécessite 2^{47} **couples chosen plaintext** (pour DES) pour obtenir des résultats corrects.

Cryptanalyse Linéaire

- Il s'agit d'une **attaque known plaintext** qui crée un **simulateur du bloc** à partir des **approximations linéaires**. En analysant un **grand nombre** de paires plaintext/ciphertexts, les **bits de la clé du simulateur** ont tendance à **coïncider** avec ceux du block cipher analysés (**calcul probabiliste**)
- Pour DES une attaque basée sur cette technique nécessite 2^{38} **known plaintexts** pour obtenir une probabilité de **10%** de deviner juste et 2^{43} **pour un 85% !**
- Il s'agit de l'**attaque analytique la plus puissante** à ce jour sur les block ciphers.

Techniques de Cryptanalyse sur les Block Ciphers (II)

- La mise en pratique des **attaques différentielles et linéaires** présente des **difficultés dans la parallélisation** des calculs par rapport à une recherche exhaustive de la clé.
- Ces deux attaques sont **très sensibles au nombre d'étapes** du block cipher: les chances de réussite augmentent **exponentiellement** au fur et à mesure que le nombre d'étapes de l'algorithme diminue.
- Une conjecture très répandue parmi les cryptographes est que ces attaques, à l'époque **inédites**, étaient **connues par les concepteurs des DES**. En particulier, le **design des S-boxes** offre une **résistance très grande** aux deux techniques.

Attaque Meet-in-the-Middle

- S'applique aux constructions du type $y := E_{K_2}(E_{K_1}(x))$. Pour DES, l'espace de clés pour cette solution serait de $\{0, 1\}^{112}$. On construit d'abord **deux listes** L_1 et L_2 de 2^{56} messages de la forme: $L_1 = E_{K_1}(x)$ et $L_2 = D_{K_2}(y)$ avec E et D les opérations d'encryption et decryption respectivement. Il faut alors **identifier des éléments qui se répètent** dans les deux listes et **vérifier notre hypothèse** avec un deuxième known plaintext. Les K_1 et K_2 associées à cette paire de known plaintexts seront (en toute vraisemblance) **les clés recherchées !**
- **Effort nécessaire** à réaliser les attaques (pour DES): 2^{57} **opérations** pour établir les deux listes + 2^{56} **blocs** de 64 bits de stockage pour mémoriser les résultats intermédiaires... **nettement inférieur** au 2^{112} estimé intuitivement...

- Ces techniques meet-in-the-middle sont aussi appliquées à la **cryptanalyse interne** des block ciphers.

💡 Révision rapide (Cryptanalyse)

Différentielle : chosen plaintext, propagation différences, probabilités sur clés, 2^{47} couples (DES).

Linéaire : known plaintext, approximations linéaires, 2^{38} - 2^{43} plaintexts (DES), attaque la plus puissante.

Meet-in-Middle : constructions composées, 2 listes 2^{56} , effort $2^{57} \ll 2^{112}$.

Sensibilité : très dépendantes du nombre de rounds.