🔐 CM2

Cryptographie symétrique · AES et modes d’opération

Module Cryptographie · Première année ingénieur

Kahoot

https://rahoot.isen-cyber.ovh

🔙 Rappel CM1

L’essentiel à retenir

🔒 Les 3 piliers de la sécurité (CIA)

- 🔒 **Confidentialité** (*Confidentiality*) — seuls les destinataires autorisés peuvent lire le message
- ✅ **Intégrité** (*Integrity*) — le message n'a pas été modifié en transit
- 🪪 **Authentification** (*Authentication*) — on sait qui a envoyé le message

Et un 4ème principe essentiel :

- 📝 **Non-répudiation** — l'expéditeur **ne peut pas nier** avoir envoyé le message (preuve)

🤔 Quel pilier est concerné ?

Situation 1 : Vous envoyez un message sur WhatsApp. Personne d'autre que votre destinataire ne peut le lire.

➡️ **Confidentialité** 🔒

Situation 2 : Vous téléchargez un logiciel. Vous vérifiez que le fichier n'a pas été corrompu avec un hash SHA-256.

➡️ **Intégrité** ✅

Situation 3 : Votre navigateur vérifie le certificat de google.com pour s'assurer que c'est bien Google et pas un imposteur.

➡️ **Authentification** 🪪

🤔 Quel pilier est concerné ? (suite)

Situation 4 : Un client signe électroniquement un contrat. Il ne peut pas dire ensuite "ce n'est pas moi qui ai signé".

➡️ **Non-répudiation** 📝

Situation 5 : Quelqu'un intercepte votre virement bancaire et change le montant de 100€ à 10 000€.

➡️ Violation de l'**intégrité** ✅ (et potentiellement de l'authentification)

Situation 6 : Votre voisin capte votre WiFi et lit vos emails.

➡️ Violation de la **confidentialité** 🔒

📝 Les fondamentaux

- 🔒 **Chiffrement** : transformer un message clair en message illisible
- 🔓 **Déchiffrement** : retrouver le clair **avec** la clé
- 🔍 **Décryptage** : retrouver le clair **sans** la clé (= casser le chiffrement)
- 🔑 **Clé** : paramètre secret qui contrôle le chiffrement
- 📏 **Espace de clés** : nombre total de clés possibles (2^n pour n bits)

"La sécurité d’un système ne doit pas dépendre du secret de l’algorithme, mais **uniquement du secret de la clé.**" 🔑

- Auguste Kerckhoffs, 1883

📊 3000 ans en un slide

Époque	Système	Faiblesse
~700 av. J.-C.	Scytale (transposition)	Peu de bâtons possibles
~50 av. J.-C.	César (substitution)	26 clés → force brute triviale
IXe siècle	Al-Kindi	**Casse** César par analyse fréquentielle
XVIe siècle	Vigenère (polyalphabétique)	"Indéchiffrable" 300 ans… puis cassé
1939-45	Enigma (rotors)	Cassé par Turing & les Polonais

➡️ **Pattern** : chaque système est cassé par la génération suivante.

⊕ Le XOR : brique fondamentale

Rappel de l'opération XOR (OU exclusif) :

```
0 ⊕ 0 = 0      "Même → 0"
0 ⊕ 1 = 1      "Différent → 1"
1 ⊕ 0 = 1
1 ⊕ 1 = 0
```

Propriété magique : le XOR est **sa propre inverse** !

```
Message    ⊕ Clé = Chiffré
Chiffré    ⊕ Clé = Message     ← on retrouve le message !
```

C'est l'opération de base de **toute la crypto moderne**. On va la retrouver partout dans AES. 🧱

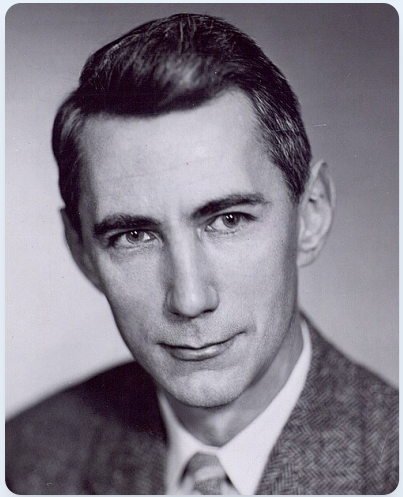
📈 Du César à AES

	César	AES
Type	Substitution simple	Chiffrement par blocs
Clé	1 nombre (0-25)	128/192/256 bits
Espace de clés	26	2^128 à 2^256
Cassable ?	⚡ Instantanément	🌌 Pas avant la fin de l'univers

Même principe : une clé transforme un message clair en message chiffré.

Complexité radicalement différente.

➡️ Comment est-on passé de l'un à l'autre ? Il y a eu une étape intermédiaire : **DES**.



🧠 Claude Shannon (1916–2001)

Le **"père de la théorie de l'information"**.

- 📡 **1948** : *A Mathematical Theory of Communication* — invente la théorie de l'information, définit le **bit** comme unité fondamentale
- 🔐 **1949** : *Communication Theory of Secrecy Systems* — pose les bases mathématiques de la cryptographie moderne
- 🎓 Sa thèse de master (1937) prouve que l'algèbre de Boole peut réaliser n'importe quel circuit logique → **fondement de l'informatique**

Un seul homme a fondé deux disciplines majeures : **la théorie de l'information** et **la cryptographie mathématique**.

🔀 Confusion et diffusion (Shannon, 1949)

Dans *Communication Theory of Secrecy Systems*, Shannon définit **deux propriétés** essentielles pour un bon chiffrement :

- 🔀 **Confusion** : rendre la relation entre la **clé** et le **chiffré** la plus complexe possible
  - → Chaque bit du chiffré dépend de **plusieurs bits de la clé**
  - → Réalisée par la **substitution** (remplacer des symboles par d'autres)
- 🌊 **Diffusion** : répartir l'influence de chaque bit du **clair** sur **tout le chiffré**
  - → Changer 1 bit du clair change ~50% des bits du chiffré
  - → Réalisée par la **transposition** (réarranger les positions)

💡 Rappel CM1 : le chiffre **ADFGVX** (1918) combinait déjà les deux ! Substitution via la grille 6×6, puis transposition par colonnes. Shannon a formalisé ce que les militaires faisaient intuitivement.

🔀 Confusion et diffusion en pratique

Principe	Opération	Effet	Dans ADFGVX
Confusion	Substitution	Masque le lien clé ↔ chiffré	Grille 6×6 (lettre → paire)
Diffusion	Transposition	Propage chaque bit partout	Réarrangement par colonnes

Rappel CM1 : ADFGVX fait d'abord une **substitution** (chaque lettre → deux symboles via la grille), puis une **transposition** (colonnes mélangées selon la clé).

Un bon chiffrement **combine les deux**. C'est exactement ce que font **DES** et **AES** : empiler des rounds qui alternent confusion et diffusion.

🏛️ DES

Data Encryption Standard (1977)

🏛️ DES : contexte

Années 1970 : le gouvernement américain a besoin d'un **standard de chiffrement**.

- 🏢 **1973** : le NIST lance un appel à propositions
- 💡 **IBM** propose un algorithme conçu par **Horst Feistel**
- 📜 **1977** : **DES** devient le standard mondial
- 📐 Bloc de **64 bits**, clé de **56 bits**, **16 rounds**

La grande idée de DES : la **structure de Feistel** — un schéma élégant qu'on retrouve dans beaucoup d'algorithmes modernes.

🔄 Structure de Feistel : le principe

Le bloc de 64 bits est **coupé en deux** : moitié gauche (**L**) et moitié droite (**R**).

À chaque round :

1. **R** passe dans une fonction **F** avec la sous-clé du round **Ki**
2. On **XOR** le résultat avec **L**
3. On **échange** L et R

```
Nouveau L = ancien R
Nouveau R = ancien L  XOR  F(ancien R, Ki)
```

On répète ça **16 fois**, avec une sous-clé différente à chaque round.

🔄 Feistel à la main (exemple simplifié)

Prenons un bloc de **8 bits** (au lieu de 64) et une seule round :

```
Bloc    : 1010 1100
L0      : 1010          R0 : 1100
Clé K1  : 0111
```

Étape 1 — Fonction F : on fait un XOR entre R0 et K1 (dans le vrai DES, F est plus complexe)

```
F(R0, K1) = 1100 XOR 0111 = 1011
```

Étape 2 — XOR avec L0 :

```
L0 XOR F = 1010 XOR 1011 = 0001
```

Étape 3 — Échange :

```
L1 = R0     = 1100
R1 = résultat = 0001
```

🔓 Feistel : et le déchiffrement ?

La magie de Feistel : pour déchiffrer, on applique **le même algorithme** avec les sous-clés **à l'envers**.

Reprenons notre exemple — on part de `1100 0001` :

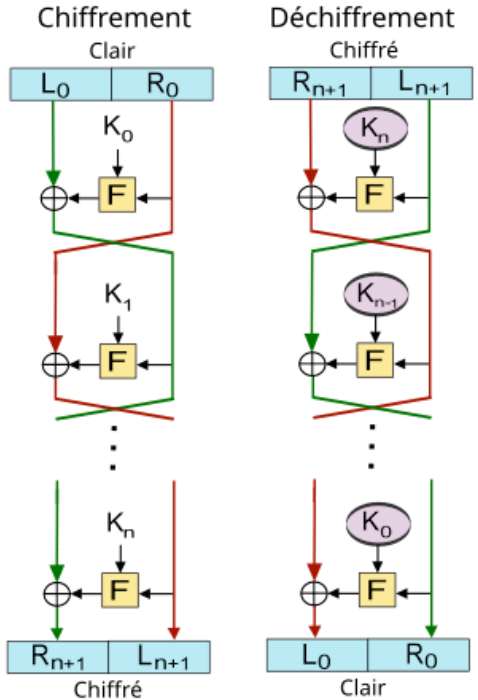
```
L1 = 1100          R1 = 0001
F(L1, K1) = 1100 XOR 0111 = 1011
R1 XOR F  = 0001 XOR 1011 = 1010

Résultat : 1010 1100  ← on retrouve le bloc original !
```

Pourquoi ça marche ? Grâce au XOR qui est sa propre inverse :

`A XOR B XOR B = A`

➡️ Un seul circuit pour chiffrer ET déchiffrer. C'est pour ça que Feistel est si élégant.

🔄 Feistel : chiffrement et déchiffrement



À gauche : chiffrement avec les clés K_0, K_1, ..., K_n. À droite : déchiffrement avec les **mêmes clés dans l'ordre inverse**.

💀 DES est cassé (1999)

Le problème de DES : sa clé de **56 bits** est trop courte.

- 🕵️ La NSA avait réduit la clé de 128 à 56 bits (pour pouvoir le casser si besoin)
- 🖥️ **1999** : l'EFF construit **Deep Crack** pour 250 000 $
- ⏱️ DES cassé en **22 heures** par force brute
- 📉 Aujourd'hui : un GPU casserait DES en **quelques minutes**

➡️ 56 bits ne suffisent plus. Il faut un remplaçant.

🔄 Les tentatives : 2DES et 3DES

Idée naturelle : appliquer DES plusieurs fois pour augmenter la sécurité.

- ❌ **2DES** (2 clés, 2 passes) : semble offrir 112 bits de sécurité, mais une attaque astucieuse (*meet-in-the-middle*) réduit ça à **~57 bits** → quasi inutile
- ✅ **3DES** (3 clés, 3 passes : Encrypt-Decrypt-Encrypt) : sécurité effective de **112 bits** → suffisant pendant ~20 ans

```
3DES : Chiffré = DES(K3, DES⁻¹(K2, DES(K1, Message)))
```

Mais 3DES est 3× plus lent et reste limité à des blocs de 64 bits.

Deprecated par le NIST en **2023**. Le monde avait besoin d'un **vrai** remplaçant → **AES**.

📏 Taille de clé et résistance

Combien de bits pour être en sécurité ?

🔢 56 bits vs 128 bits vs 256 bits

Taille	Nombre de clés	Temps estimé (1 milliard/s)	Algo
56 bits	7.2 × 10^16	⏱️ ~2.3 ans	DES
112 bits	5.2 × 10^33	🌌 ~10^17 ans	3DES
128 bits	3.4 × 10^38	🌌 ~10^22 ans	AES-128
256 bits	1.2 × 10^77	🌌🌌 ~10^60 ans	AES-256

L'âge de l'univers : ~1.4 × 10^10 ans.

AES-128 demanderait 10^12 fois l'âge de l'univers pour être cassé par force brute. 🤯

🔢 Chaque bit double la difficulté

Ajouter **1 bit** à la clé → **2× plus** de clés à tester.

```
56 bits  →  72 quadrillions de clés
57 bits  →  144 quadrillions        (×2)
58 bits  →  288 quadrillions        (×2)
...
128 bits → 340 undécillions
```

De 56 à 128 bits = 72 bits de plus = **2^72 fois plus de clés**.

2^72 ≈ 4.7 × 10^21 = **4 700 milliards de milliards** de fois plus difficile.

C'est pour ça que la NSA voulait 56 et pas 128 : **72 bits de différence, c'est un gouffre**. 🕳️

Et les ordinateurs quantiques ?

Algorithme de Grover : réduit la force brute à la **racine carrée**.

Clé classique	Sécurité post-quantique
AES-128 (2^128)	2^64 → ⚠️ potentiellement cassable
AES-256 (2^256)	2^128 → ✅ toujours sûr

Recommandation : utiliser AES-256 pour être **quantum-resistant** 🚀

(Les ordinateurs quantiques capables de ça n'existent pas encore, mais on anticipe.)

🏆 AES : Advanced Encryption Standard

Le compétition (1997-2001)

🏆 La compétition AES

1997 : le NIST lance un appel mondial pour remplacer DES.

- 📬 **15 candidats** soumis par des équipes du monde entier
- 🔬 Critères : sécurité, performance, simplicité, flexibilité
- 🥇 **5 finalistes** : Rijndael, Serpent, Twofish, RC6, MARS
- 🥇 **2001** : **Rijndael** est choisi → devient AES

Conçu par deux cryptographes belges : **Joan Daemen** et **Vincent Rijmen**.

🇧🇪 Rijndael = **Rijn**men + **Dae**men

🤔 Pourquoi Rijndael a gagné ?

- 🔒 **Sécurité** : aucune attaque significative trouvée pendant la compétition
- ⚡ **Performance** : très rapide en software ET hardware
- 🧩 **Simplicité** : design élégant et analysable
- 📐 **Flexibilité** : supporte 128, 192 et 256 bits de clé
- 🛡️ **Transparence** : entièrement public, analysé par des centaines de cryptographes

Contrairement à DES, **pas d'intervention de la NSA** cette fois. Processus ouvert et transparent.

⚙️ AES : structure interne

Comment ça marche concrètement

📦 AES : vue d'ensemble

Chiffrement par blocs symétrique

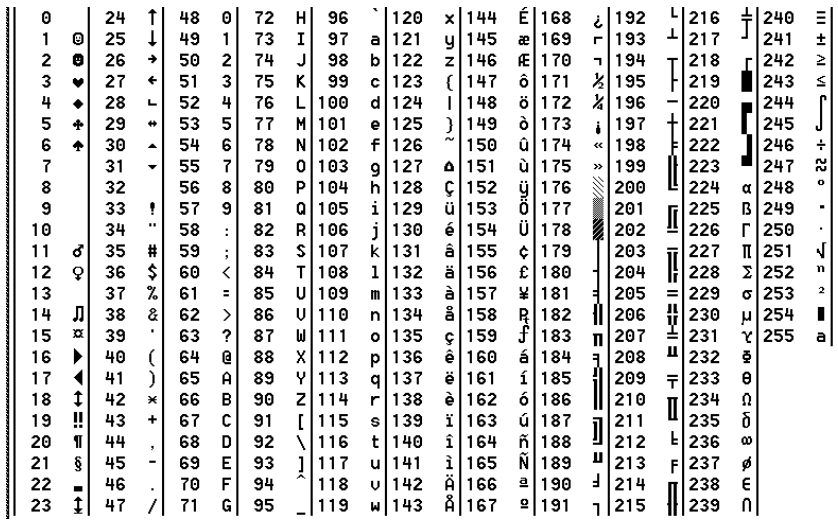
- 📐 Taille de bloc : **128 bits** (16 octets), toujours
- 🔑 Tailles de clé : **128**, **192** ou **256** bits
- 🔄 Nombre de rounds :
  - 128 bits → **10 rounds**
  - 192 bits → **12 rounds**
  - 256 bits → **14 rounds**

Chaque round applique **4 opérations** qui transforment le bloc.

Plus la clé est longue → plus de rounds → plus de sécurité.

📋 Rappel : la table ASCII

Chaque caractère est codé sur **1 octet** (8 bits) = une valeur **hexadécimale** de 00 à FF.



AES travaille sur des **octets** (valeurs 0x00 à 0xFF).

Quand on chiffre `"Hello"`, AES voit : `48 65 6C 6C 6F`

AES : l'état (State)

Le bloc de 128 bits (16 octets) est organisé en **matrice 4×4** d'octets :

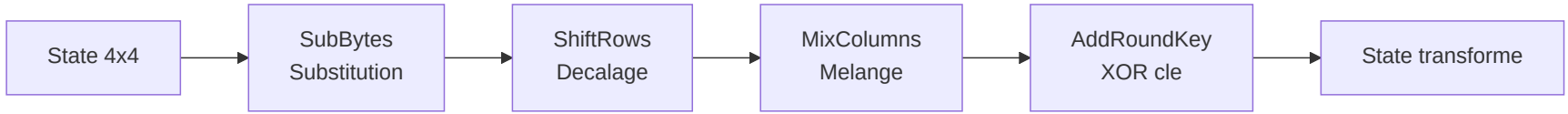
```
Message : "Hello AES World!" (16 caractères = 16 octets)
H  e  l  l  o  (sp)  A  E  S  (sp)  W  o  r  l  d  !
48 65 6C 6C 6F  20   41 45 53  20   57 6F 72 6C 64 21
```

La matrice **State** est remplie **colonne par colonne** :

```
| 48 (H)  | 6F (o)  | 53 (S)  | 72 (r)  |
| 65 (e)  | 20 ( )  | 20 ( )  | 6C (l)  |
| 6C (l)  | 41 (A)  | 57 (W)  | 64 (d)  |
| 6C (l)  | 45 (E)  | 6F (o)  | 21 (!)  |
```

Chaque case = **1 octet** (8 bits, valeur 0x00 à 0xFF). La matrice est toujours **pleine** (16 cases pour 16 octets).

🔄 AES : un round



4 opérations par round, répétées **10/12/14 fois**.

Le dernier round **n'a pas de MixColumns** (raison : n'ajoute pas de sécurité au dernier tour et facilite le déchiffrement).

🎲 SubBytes

La substitution non-linéaire

🎲 SubBytes : principe

Chaque octet du State est **remplacé** par un autre octet via une **table de substitution** (S-Box).

```
Entrée : 0x53 → Sortie : 0xED
Entrée : 0x6C → Sortie : 0x50
Entrée : 0x00 → Sortie : 0x63
```

- La S-Box est une **table fixe** de 256 entrées (une pour chaque valeur d'octet 0x00 à 0xFF)
- La même S-Box est utilisée pour **tous les rounds**, **toutes les clés**
- Elle est **publique** (principe de Kerckhoffs !)

La S-Box AES complète

On lit la table : **ligne = premier chiffre hex**, **colonne = deuxième chiffre hex**.

	0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	0F
0	63	7C	77	7B	F2	6B	6F	C5	30	1	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	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
40	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
50	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
60	D0	EF	AA	FB	43	4D	33	85	45	F9	2	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	6	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	8
C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D0	70	3E	B5	66	48	3	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

Exemple : octet **0x53** → Ligne **50**, colonne **3** → résultat : **0xED**

SubBytes : exemple complet

Avant SubBytes :

```
| 48 (H)  | 6F (o)  | 53 (S)  | 72 (r)  |
| 65 (e)  | 20 ( )  | 20 ( )  | 6C (l)  |
| 6C (l)  | 41 (A)  | 57 (W)  | 64 (d)  |
| 6C (l)  | 45 (E)  | 6F (o)  | 21 (!)  |
```

Après SubBytes (chaque octet remplacé via la S-Box) :

```
| 52      | A8      | ED      | 40      |
| 4D      | B7      | B7      | 50      |
| 50      | 83      | 5B      | 43      |
| 50      | 6E      | A8      | FD      |
```

Chaque octet est transformé **indépendamment**. C'est de la **confusion** (Shannon).

🎲 Pourquoi cette S-Box ?

La S-Box n'est **pas aléatoire**. Elle a été construite mathématiquement pour :

- ❌ **Pas de point fixe** : aucun octet ne se transforme en lui-même ($S[x] \neq x$)
- ❌ **Pas d'anti-point fixe** : $S[x] \neq \bar{x}$ (complément)
- 🔀 **Non-linéarité maximale** : résistante aux attaques linéaires et différentielles
- ✅ **Inversible** : pour pouvoir déchiffrer, il faut une S-Box inverse

C'est la partie qui rend AES **résistant aux attaques mathématiques**.

Sans la S-Box, AES serait juste un système linéaire → cassable facilement.

↔️ ShiftRows

Le décalage des lignes

↔️ ShiftRows : principe

Chaque **ligne** de la matrice est décalée vers la gauche d'un nombre de positions différent.

```
Ligne 0 : pas de décalage    (shift 0)
Ligne 1 : décalage de 1      (shift 1)
Ligne 2 : décalage de 2      (shift 2)
Ligne 3 : décalage de 3      (shift 3)
```

Objectif : mélanger les octets entre les colonnes.

Avant ShiftRows, chaque colonne est traitée indépendamment. Après, les octets d'une colonne viennent de **colonnes différentes → diffusion**.

↔️ ShiftRows : exemple

Avant ShiftRows (= après SubBytes) :

```
| 52  A8  ED  40 |   ← Ligne 0 : pas de décalage
| 4D  B7  B7  50 |   ← Ligne 1 : shift 1
| 50  83  5B  43 |   ← Ligne 2 : shift 2
| 50  6E  A8  FD |   ← Ligne 3 : shift 3
```

Après ShiftRows :

```
| 52  A8  ED  40 |   ← inchangée
| B7  B7  50  4D |   ← 4D passe à la fin
| 5B  43  50  83 |   ← 50,83 passent à la fin
| FD  50  6E  A8 |   ← 50,6E,A8 passent à la fin
```

Simple mais efficace : les octets sont maintenant **redistribués** entre les colonnes. 🔀

MixColumns

Le mélange des colonnes

🧮 MixColumns : principe

Chaque **colonne** (4 octets) est transformée par une **multiplication matricielle**.

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

La matrice est toujours la même (constante fixe d'AES).

Les opérations se font dans **GF(2^8)** — le **corps de Galois** à 256 éléments :

- L'**addition** = XOR (pas de retenue, pas de dépassement)
- La **multiplication** = multiplication de polynômes modulo un polynôme irréductible

Concrètement : ça garantit que chaque résultat reste **un octet** (0x00–0xFF), et que chaque opération est **inversible** — indispensable pour pouvoir déchiffrer.

🧮 MixColumns : intuition

Ce qu'il faut retenir (sans les maths) :

- Chaque octet de sortie dépend des **4 octets d'entrée** de la colonne
- Si **1 octet** change en entrée → les **4 octets** de la colonne changent en sortie
- C'est de la **diffusion** maximale au sein d'une colonne

Exemple réel — on prend la 1ère colonne et on applique MixColumns :

```
Colonne avant : | 52 | 83 | 40 | 63 |
Colonne après : | 19 | EC | F4 | F3 |
```

Changeons juste 52 → 53 (1 seul bit de différence) :

```
Colonne avant : | 53 | 83 | 40 | 63 |
Colonne après : | 1B | ED | F5 | F0 |  ← les 4 octets changent !
```

🧮 MixColumns : pourquoi ces coefficients ?

```
| 2  3  1  1 |
| 1  2  3  1 |
| 1  1  2  3 |
| 3  1  1  2 |
```

- **Coefficients 1, 2, 3** : les plus petits possibles → calcul rapide
- **Matrice circulante** : chaque ligne est un décalage de la précédente → structure régulière
- **Inversible** : il existe une matrice inverse (nécessaire pour le déchiffrement)
- **Diffusion optimale** : le minimum de coefficients nécessaire pour que chaque sortie dépende de toutes les entrées

⊕ AddRoundKey

L’injection de la clé

⊕ AddRoundKey : principe

Le State est **XORé** bit à bit avec une **sous-clé** de 128 bits (dérivée de la clé principale).

```
State    ⊕ Sous-clé du round = State transformé
```

Exemple (un seul octet) :

```
State    : 0x52 = 0101 0010
Sous-clé : 0xA3 = 1010 0011
           ⊕
Résultat : 0xF1 = 1111 0001
```

C'est l'opération qui **injecte la clé** 🔑

Sans AddRoundKey, le chiffrement serait identique quelle que soit la clé → inutile !

🔑 Key Schedule (expansion de clé)

La clé principale (128/192/256 bits) est **étendue** en plusieurs sous-clés :

- AES-128 : 1 clé → **11 sous-clés** de 128 bits (1 initiale + 10 rounds)
- AES-192 : 1 clé → **13 sous-clés**
- AES-256 : 1 clé → **15 sous-clés**

Chaque round utilise une sous-clé différente.

Le Key Schedule utilise aussi la S-Box et des constantes (Rcon) pour dériver les sous-clés. Chaque sous-clé dépend de la clé originale de manière complexe.

📋 AES : résumé des 4 opérations

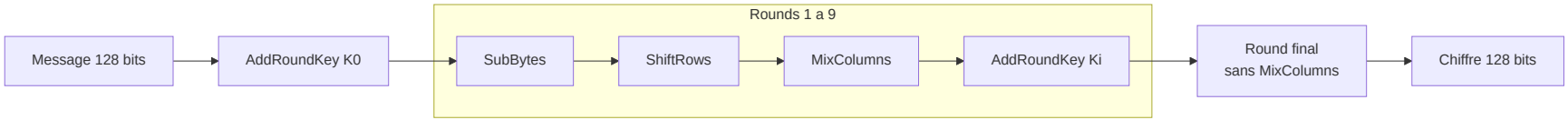
Opération	Ce qu'elle fait	Propriété (Shannon)
🎲 **SubBytes**	Remplace chaque octet via S-Box	**Confusion**
↔️ **ShiftRows**	Décale les lignes	**Diffusion**
🧮 **MixColumns**	Mélange chaque colonne (matrice)	**Diffusion**
⊕ **AddRoundKey**	XOR avec la sous-clé du round	**Injection de clé**

Confusion = rendre la relation clé/chiffré complexe (S-Box non-linéaire)

Diffusion = un changement d'1 bit se propage partout (ShiftRows + MixColumns)

Les deux ensemble → **effet avalanche** : 1 bit changé dans le clair → ~50% des bits du chiffré changent. 🌊

🔄 AES complet : vue d'ensemble



🔓 Et le déchiffrement ?

AES est **inversible** : chaque opération a son inverse.

Chiffrement	Déchiffrement
SubBytes (S-Box)	InvSubBytes (S-Box inverse)
ShiftRows (décalage gauche)	InvShiftRows (décalage droite)
MixColumns (matrice M)	InvMixColumns (matrice M^{-1})
AddRoundKey (⊕ clé)	AddRoundKey (⊕ clé) ← même opération !

Le déchiffrement applique les opérations **inverses** dans l'**ordre inverse**, avec les sous-clés à l'envers.

Note : AddRoundKey est son propre inverse car **A ⊕ K ⊕ K = A** (propriété du XOR).

🌊 L'effet avalanche en action

Deux messages qui diffèrent d'**un seul caractère** — testons avec `openssl` :

```
KEY="0123456789abcdef0123456789abcdef"
IV="00000000000000000000000000000000"

echo -n "Hello AES World!" | openssl enc -aes-128-cbc -K $KEY -iv $IV -nopad | xxd -p
# → ed7df692101a6a6270cd5b00f7c5768d

echo -n "Hello AES World?" | openssl enc -aes-128-cbc -K $KEY -iv $IV -nopad | xxd -p
# → e5d1363d5f8cfdd1be49dd33577905fb
```

Même clé, même IV, un seul `!` → `?` et le chiffré est **totalement différent**.

Aucun octet en commun — c'est **l'effet avalanche** 🌊

C'est la combinaison SubBytes + ShiftRows + MixColumns sur **10 rounds** qui crée cette propagation.

➡️ Impossible de deviner quoi que ce soit sur le clair à partir du chiffré.

🏗️ Résumé : DES vs 2DES vs 3DES vs AES

	DES	2DES	3DES	AES
Année	1977	—	~1998	2001
Bloc	64 bits	64 bits	64 bits	**128 bits**
Clé	56 bits	2×56 bits	3×56 bits	128/192/256
Sécurité effective	56 bits	**~57 bits ❌**	112 bits	128-256 bits
Statut	❌ Cassé	❌ Inutile	⚠️ Deprecated	✅ **Standard**

AES est **le standard mondial** depuis 2001. Utilisé partout : HTTPS, WiFi, disques chiffrés, messageries... 🌍

Padding PKCS7

❓ Le problème du padding

AES chiffre des blocs de **128 bits** (16 octets).

Que faire si le message n'est pas un multiple de 16 octets ?

```
Message  : "Hello" → 5 octets
Bloc AES : 16 octets
Manque   : 11 octets 🤷
```

➡️ Il faut **compléter** le dernier bloc.

✏️ PKCS7 : la règle

Règle : compléter avec des octets dont la **valeur = nombre d'octets ajoutés**.

```
Message (5 octets)  : H  e  l  l  o
Padding (11 octets) :                0B 0B 0B 0B 0B 0B 0B 0B 0B 0B 0B
```

Autre exemple — message de 13 octets :

```
Message (13 octets) : H  e  l  l  o  ,     W  o  r  l  d  !
Padding (3 octets)  :                                    03 03 03
```

Cas particulier : message de 16 octets (= 1 bloc exact) → on ajoute un **bloc entier** de padding (16 × `0x10`).

Pourquoi ? Pour que le déchiffrement sache toujours combien retirer. ✂️

✏️ PKCS7 : déchiffrement

Au déchiffrement, c'est simple :

1. Déchiffrer le dernier bloc
2. Lire la **valeur du dernier octet** → c'est le nombre d'octets de padding
3. Vérifier que les N derniers octets ont bien tous la même valeur
4. Les retirer ✂️

Exemple :

```
Bloc déchiffré : ... 48 65 6C 6C 6F 03 03 03
                                    ↑  ↑  ↑
                     Dernier octet = 03 → retirer 3 octets
                     Message : "Hello"
```

🔀 Modes d’opération

Comment enchaîner les blocs

🧩 Le problème des blocs

AES chiffre **un bloc de 16 octets** à la fois.

Un message fait souvent **plusieurs blocs**.

Exemple : un email de 1000 caractères = **63 blocs** à chiffrer.

Comment enchaîner les blocs ? ➡️ C'est le **mode d'opération**.

⚠️ Le choix du mode a un impact **énorme** sur la sécurité.

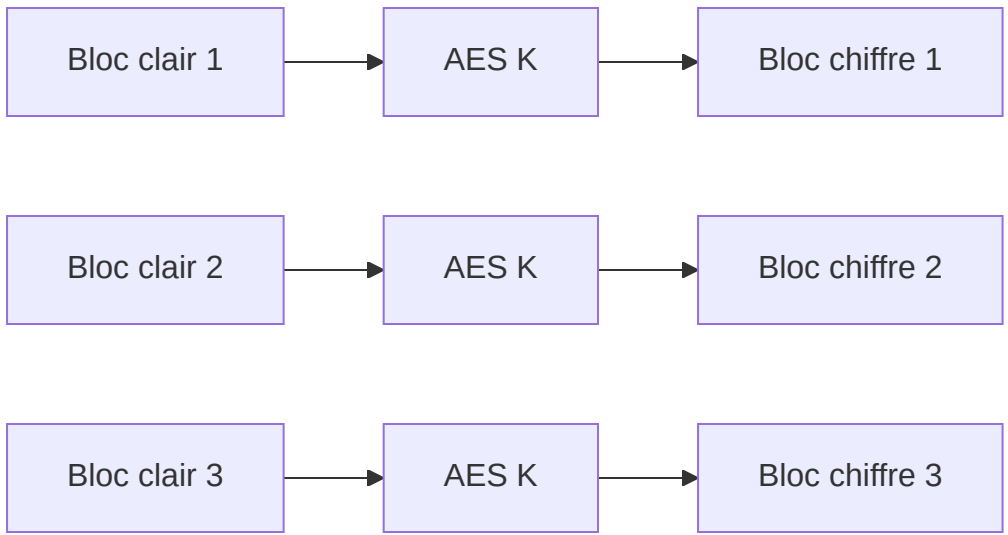
AES est sûr. Mais **AES mal utilisé** peut être dangereux.

📦 Mode ECB

Electronic Code Book

📦 Mode ECB (Electronic Code Book)

Chaque bloc est chiffré **indépendamment** avec la même clé.



Simple. Rapide. Parallélisable. **Mais…** 😬

⚠️ ECB : le problème fondamental

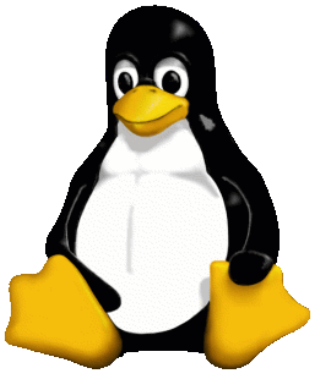
Deux blocs clairs **identiques** → deux blocs chiffrés **identiques**.

```
Bloc clair 1 : AAAAAAAAAAAAAAAA → Bloc chiffré : 7f3e2a...
Bloc clair 2 : AAAAAAAAAAAAAAAA → Bloc chiffré : 7f3e2a... ← identique ! 😱
Bloc clair 3 : BBBBBBBBBBBBBBBB → Bloc chiffré : 9c4d1b...
```

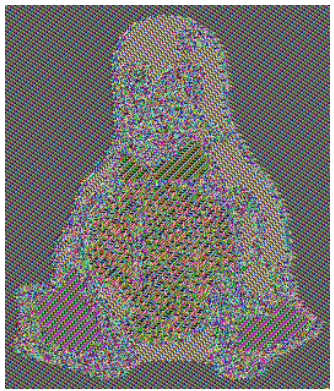
C'est une fonction déterministe : même entrée + même clé = même sortie. Toujours.

ECB **ne cache pas les patterns** du message original.

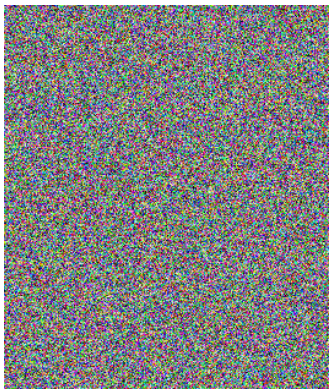
🐧 L’effet pingouin : ECB



🖼️ Original



⚠️ Chiffré ECB



✅ Chiffré CBC

Les zones de même couleur produisent les mêmes blocs chiffrés !

➡️ **ECB est dangereux. Ne jamais l’utiliser.** ❌

🐧 Pourquoi ça se voit sur une image ?

Une image BMP stocke les pixels **ligne par ligne**, sans compression.

- Le fond blanc du pingouin = des **milliers de blocs identiques** (pixels blancs)
- Le ventre blanc = aussi des blocs identiques
- En ECB : tous ces blocs blancs → **même chiffré**
- Les couleurs changent… mais les **zones de même couleur** restent visibles

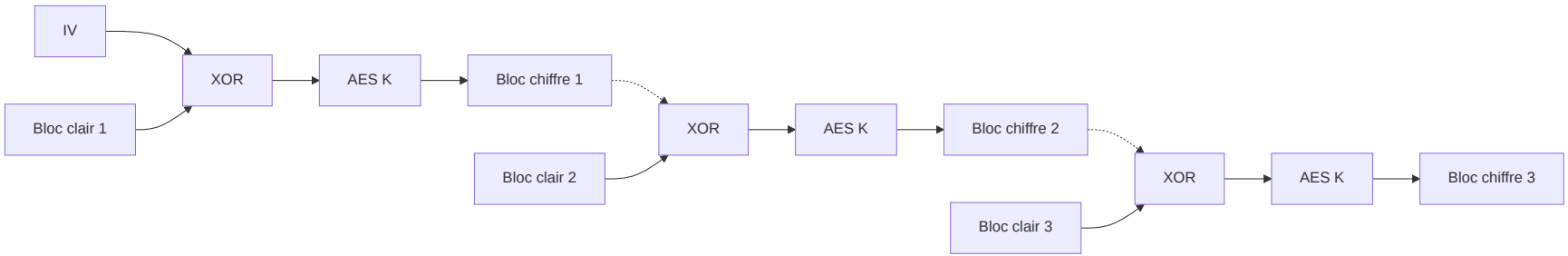
Le chiffrement ECB préserve la structure de l'image.

C'est comme si on avait juste changé la palette de couleurs — la silhouette reste. 🎨

🔗 Mode CBC
Cipher Block Chaining

🔗 Mode CBC (Cipher Block Chaining)

Chaque bloc clair est **XORé** avec le bloc chiffré précédent avant d'être chiffré.



🎲 CBC : le vecteur d'initialisation (IV)

Le **premier bloc** n'a pas de bloc chiffré précédent.

➡️ On utilise un **IV** (Initialization Vector) aléatoire de 16 octets.

Propriétés :

- 🔄 L'IV est **différent** à chaque chiffrement
- 📢 L'IV n'est **pas secret** : il est transmis en clair avec le chiffré
- 🎲 L'IV doit être **aléatoire et imprévisible** (sinon, attaques possibles)
- 📐 L'IV fait **16 octets** (= taille d'un bloc AES)

✅ CBC : pourquoi c'est mieux

Deux blocs clairs **identiques** → deux blocs chiffrés **différents** !

Pourquoi ?

Bloc 1 : $\text{Clair}_1 \oplus \text{IV}$ → les entrées d'AES sont différentes (IV aléatoire)

Bloc 2 : $\text{Clair}_2 \oplus \text{Chiffré}_1$ → même si $\text{Clair}_2 = \text{Clair}_1$, $\text{Chiffré}_1 \neq \text{IV}$

Le chaînage **casse les patterns**.

🐧 L'effet pingouin en CBC → **bruit aléatoire**, silhouette invisible. ✅

🔄 CBC : le chaînage en détail

```
Bloc 1 : C₁ = AES(K, P₁ ⊕ IV)
Bloc 2 : C₂ = AES(K, P₂ ⊕ C₁)
Bloc 3 : C₃ = AES(K, P₃ ⊕ C₂)
...
```

Déchiffrement :

```
P₁ = AES⁻¹(K, C₁) ⊕ IV
P₂ = AES⁻¹(K, C₂) ⊕ C₁
P₃ = AES⁻¹(K, C₃) ⊕ C₂
...
```

Remarque : le déchiffrement de chaque bloc ne dépend que du bloc chiffré précédent → **parallélisable** au déchiffrement (mais pas au chiffrement).

⚖️ ECB vs CBC : résumé

	ECB	CBC
Blocs identiques	❌ Même chiffré	✅ Chiffré différent
Patterns	❌ Visibles	✅ Cachés
IV nécessaire	Non	Oui (aléatoire)
Parallélisable (chiffrement)	✅ Oui	❌ Non
Parallélisable (déchiffrement)	✅ Oui	✅ Oui
Sécurité	❌ **Dangereux**	✅ **Sûr**

En pratique : on n'utilise **jamais** ECB pour chiffrer des données réelles.

🔓 Attaque par clair connu sur le XOR

Pourquoi le XOR seul ne suffit pas

🔑 Le chiffrement XOR simple

L'idée la plus simple : chiffrer en faisant un **XOR** entre le message et la clé (répétée).

```
Message : H  e  l  l  o        (en hex : 48 65 6C 6C 6F)
Clé     : S  E  C              (en hex : 53 45 43)
```

On répète la clé pour couvrir le message, puis on XOR octet par octet :

```
Message : 48  65  6C  6C  6F
Clé     : 53  45  43  53  45   (SEC répété)
          XOR XOR XOR XOR XOR
Chiffré : 1B  20  2F  3F  2A
```

Ça a l'air sûr… **mais si l'attaquant connaît un bout du message clair ?**

🎯 Attaque par clair connu (*known-plaintext*)

Scénario : Eve intercepte le chiffré et sait que le message commence par `"Hello"` (par exemple, un email commence toujours par un header connu).

Elle connaît le **clair** et le **chiffré** → elle peut retrouver la **clé** !

```
Chiffré connu : 1B  20  2F  3F  2A
Clair connu   : 48  65  6C  6C  6F    ("Hello")
                XOR XOR XOR XOR XOR
Clé retrouvée: 53  45  43  53  45
```

En ASCII : `53 45 43` = **"SEC"** → la clé est retrouvée !

Pourquoi ça marche ? Car `A XOR B = C` implique `C XOR A = B`

Si on connaît 2 des 3 valeurs, on retrouve la 3ème.

⚠️ Leçons de cette attaque

- Le **XOR seul n'est pas un chiffrement sûr** — un seul morceau de clair connu casse tout
- La clé est **réutilisée** à chaque répétition → même faiblesse que Vigenère !
- En pratique, les clairs partiels sont souvent **prévisibles** : headers HTTP, en-têtes de fichiers, signatures de formats…

C'est pour ça qu'AES existe : il ne fait pas juste un XOR, il ajoute de la **confusion** (S-Box) et de la **diffusion** (ShiftRows, MixColumns) sur **10+ rounds**.

Même si l'attaquant connaît le clair et le chiffré, il **ne peut pas remonter à la clé**.

📂 Des clairs connus partout : magic numbers

Presque tous les formats de fichiers commencent par des **octets prévisibles** :

Format	Premiers octets (hex)	En ASCII
PDF	`25 50 44 46`	`%PDF`
PNG	`89 50 4E 47 0D 0A 1A 0A`	`.PNG....`
JPEG	`FF D8 FF`	
ZIP	`50 4B 03 04`	`PK..`
ELF (Linux)	`7F 45 4C 46`	`.ELF`
Script bash	`23 21 2F 62 69 6E`	`#!/bin`
HTML	`3C 68 74 6D 6C`	

🏆 Le chiffre de Vernam (One-Time Pad)

Inventé par **Gilbert Vernam** en 1917, perfectionné par le renseignement militaire.

Et si la clé faisait **la même taille que le message**, était **aléatoire** et **jetable** ?

- 🔑 Clé **aussi longue** que le message
- 🎲 Clé **parfaitement aléatoire** (pas un mot, pas un pattern)
- 🗑️ Clé **jetable** : utilisée une seule fois, puis détruite

C'est le **masque jetable** (*One-Time Pad*) — prouvé **mathématiquement incassable** par Shannon en 1949.

L'attaque par clair connu ne marche plus : la clé retrouvée ne sert à rien pour le prochain message.

Utilisé en vrai : le téléphone rouge Moscou–Washington (1963) utilisait un chiffre de Vernam.

Problème : il faut transmettre une clé aussi longue que le message… de manière sécurisée. Inutilisable à grande échelle.

🧪 Exercice : retrouvez la clé !

Un serveur web chiffre ses pages avec un simple XOR à clé répétée.

Vous interceptez le chiffré (en hex) :

```
7F 3A 2D 3D 38 71 7F 3A 68 6E 1D 1C 06 1C 65 7F 3C 7E 7D 6E 76 38 20 22 2F 6C
```

Vous savez que la page web commence par `<html>`.

Indice : `Clé = Chiffré XOR Clair`. Commencez par les 6 premiers octets :

```
7F XOR 3C('<') = ?     3A XOR 68('h') = ?     2D XOR 74('t') = ?
3D XOR 6D('m') = ?     38 XOR 6C('l') = ?     71 XOR 3E('>') = ?
```

Réponse :

```
7F XOR 3C = 43 ('C')   3A XOR 68 = 52 ('R')   2D XOR 74 = 59 ('Y')
3D XOR 6D = 50 ('P')   38 XOR 6C = 54 ('T')   71 XOR 3E = 4F ('O')
```

La clé est **CRYPTO** — retrouvée avec seulement 6 octets de clair connu !

🧪 Maintenant on déchiffre tout !

On connaît la clé **CRYPTO** — on la répète sous le chiffré et on XOR :

```
Chiffré : 7F 3A 2D 3D 38 71  7F 3A 68 6E 1D 1C  06 1C 65 7F 3C 7E  7D 6E 76 38 20 22  2F 6C
Clé     :  C  R  Y  P  T  O   C  R  Y  P  T  O   C  R  Y  P  T  O   C  R  Y  P  T  O   C  R
        : 43 52 59 50 54 4F  43 52 59 50 54 4F  43 52 59 50 54 4F  43 52 59 50 54 4F  43 52
```

```
Clair   : 3C 68 74 6D 6C 3E  3C 68 31 3E 49 53  45 4E 3C 2F 68 31  3E 3C 2F 68 74 6D  6C 3E
        :  <  h  t  m  l  >   <  h  1  >  I  S   E  N  <  /  h  1   >  <  /  h  t  m   l  >
```

Le message déchiffré : `<html><h1>ISEN</h1></html>`

Avec 6 octets de clair connu, on a retrouvé la clé ET déchiffré **tout** le message. 🎯

🤝 Le problème de l’échange de clé

👫 Alice et Bob

Alice et Bob veulent communiquer de manière **sécurisée**.

Ils ont AES : un algorithme de chiffrement **excellent**. ✅

Mais pour utiliser AES, ils doivent partager une **clé secrète**. 🔑

Comment ? 🤔

❌ Option 1 : envoyer la clé sur le réseau

Si Eve intercepte la clé → elle peut **tout déchiffrer**. 💀

❌ Option 2 : se rencontrer physiquement

👫 Alice et Bob se retrouvent dans un café et échangent une clé USB.

Problèmes :

- 🌍 **Pas scalable** : un serveur web a des millions de visiteurs
- ✈️ **Pas pratique** : Alice est à Paris, Bob est à Tokyo
- 💀 **Un seul échange compromis** → tout est compromis
- 🔄 **Changement de clé** : il faudrait se revoir à chaque fois

🤔 Le paradoxe

Pour communiquer de manière sécurisée, il faut une **clé partagée**.

Pour partager une clé de manière sécurisée, il faut un **canal sécurisé**.

Pour avoir un canal sécurisé, il faut une **clé partagée**...

C'est un cercle vicieux. 🔄

C'est **le** problème fondamental de la cryptographie symétrique.

"Comment Alice et Bob peuvent-ils se mettre d’accord sur une clé secrète **sans jamais se rencontrer** et **sans qu’Eve puisse la connaître** ? 🤔"

Réponse au CM3…

📝 À retenir

- 🏛️ **DES** (56 bits) cassé en 1999 — compromis politique NSA
- ❌ **2DES** : inutile → attaque meet-in-the-middle (sécurité ~57 bits)
- ⏳ **3DES** (112 bits effectifs) : patch temporaire, deprecated en 2023
- 🏆 **AES** : compétition ouverte, 4 opérations (SubBytes, ShiftRows, MixColumns, AddRoundKey)
- 🎲 **SubBytes** = confusion (S-Box), **ShiftRows + MixColumns** = diffusion
- ❌ **ECB est dangereux** : les patterns sont visibles (🐧 effet pingouin)
- ✅ **CBC est sûr** : chaînage + IV aléatoire cache les patterns
- 🤝 Le chiffrement symétrique a un problème : **l'échange de clé** → CM3

🧪 TD2 : AES en pratique

Vous allez :

- 📦 Implémenter le chiffrement AES en **ECB** et **CBC**
- 🐧 Voir l'**effet pingouin** sur une image BMP
- 🔍 Comprendre concrètement pourquoi ECB est dangereux
- ⚖️ Comparer ECB et CBC sur des blocs identiques

❓ Questions ?

CM2 · Cryptographie symétrique