Université IBN TOFAIL – Kénitra
ECOLE NATIONALE DES SCIENCES APPLIQUEES

Cours Sécurité des Systèmes Embarqués et IoT

CH2: Cryptographie et Chiffrement

Par:

Hassan EL FADIL

elfadil.h@gmail.com

1. Cryptographie vs Chiffrement

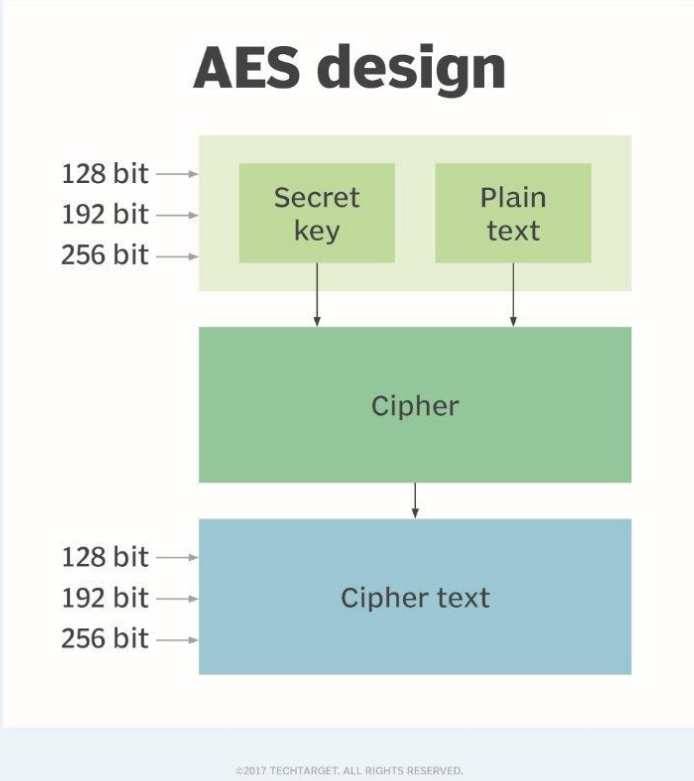
- ***Cryptographie :***
  - Science qui étudie les techniques pour sécuriser l'information (confidentialité, intégrité, disponibilité, authenticité et non-répudiation): **CIAAN** (Confidentiality, Integrity, Availability, Authentication, Non-repudiation).
  - Concept global qui inclut le chiffrement, la signature, l'authentification, etc.
- ***Chiffrement :***
  - Technique de transformation d'un message en un format illisible (**chiffré**) pour garantir la confidentialité.
  - Sous-domaine de la cryptographie.
- ***Exemple :***
  - **AES, RSA,** → Algorithmes de **chiffrement**.
  - **SHA-256, HMAC, signatures numériques** → Techniques cryptographiques qui ne font pas de chiffrement mais assurent intégrité et authentification.

2. Chiffrement Symétrique

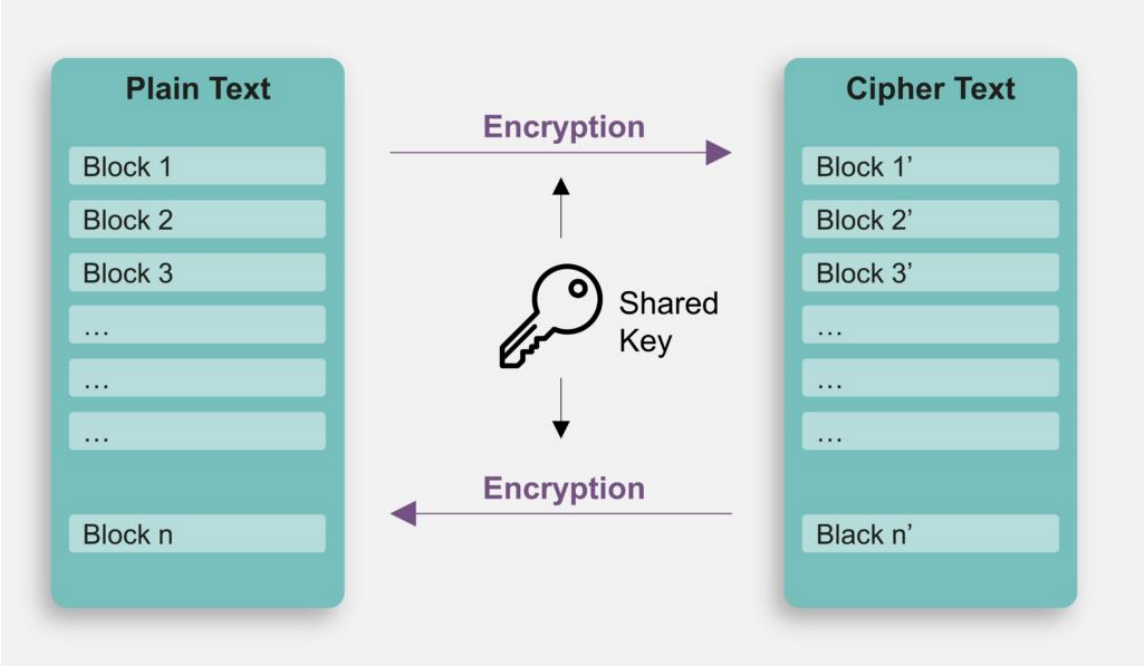
2.1. Définition de l'AES :

- **Définition** : AES (Advanced Encryption Standard) est un algorithme de chiffrement symétrique adopté comme standard par le NIST (National Institute of Standards and Technology) en 2001.
- Il est utilisé pour sécuriser les données grâce à une clé unique partagée entre l'émetteur et le récepteur (symétrique).

Texte en clair (**plaintext**)
Texte chiffré (**ciphertext**)



- **Clé unique** : Une seule clé est utilisée pour chiffrer (transformer les données en un format illisible) et déchiffrer (retrouver les données originales).
- **Blocs de données** : AES travaille sur des blocs de 128 bits (**16 octets**), mais les clés peuvent avoir une longueur de 128, 192, ou 256 bits.
- **Cycles d'opérations** : L'algorithme effectue des cycles d'opérations (appelés "rounds"), qui incluent substitution, permutation, mélange, et ajout de clé.
- **Résistance aux attaques** : Grâce à sa conception robuste, AES est résistant aux attaques par force brute et cryptanalyse linéaire.

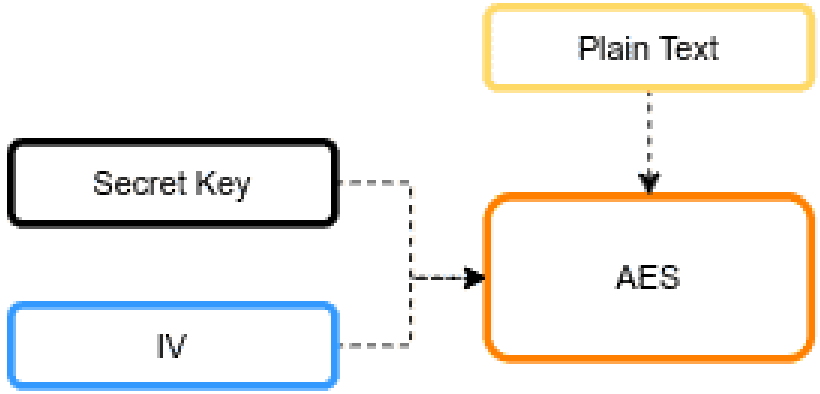


2.2. Modes de Chiffrement AES :

Mode	Acronyme	Description	Sécurité	Applications
Electronic Codebook	**ECB**	Chaque bloc est chiffré indépendamment avec la même clé.	Faible	Anciennes applications, données non sensibles ou courtes.
Cipher Block Chaining	**CBC**	Chaque bloc est XORé avec le bloc précédent avant chiffrement, utilise un IV pour le premier bloc.	Élevée	Chiffrement de fichiers, bases de données, anciens protocoles TLS.
Cipher Feedback	**CFB**	Fonctionne comme un flux : XOR du texte clair avec la sortie chiffrée précédente, nécessite un IV unique.	Moyenne	Protocoles sécurisés en temps réel : SSH, IPsec.
Output Feedback	**OFB**	Utilise la sortie AES précédente comme entrée pour générer un flux pseudo-aléatoire.	Moyenne	Protocoles sensibles aux erreurs, télécommunications.
Counter	**CTR**	Utilise un compteur incrémenté pour chaque bloc, XORé avec le texte clair.	Élevée	Protocoles sécurisés modernes : TLS, HTTPS, VPN, IPsec.

2.3. Vecteur d'Initialisation AES (IV):

- Un vecteur d'initialisation (**IV**) est une séquence de bits utilisée pour initialiser un algorithme de chiffrement, généralement dans les modes de chiffrement par blocs (comme **CBC**, **CFB**, **OFB**).
- L'IV garantit que même si le même texte en clair est chiffré plusieurs fois avec la même clé, les résultats seront différents à chaque fois, assurant ainsi la **sécurité** des données.

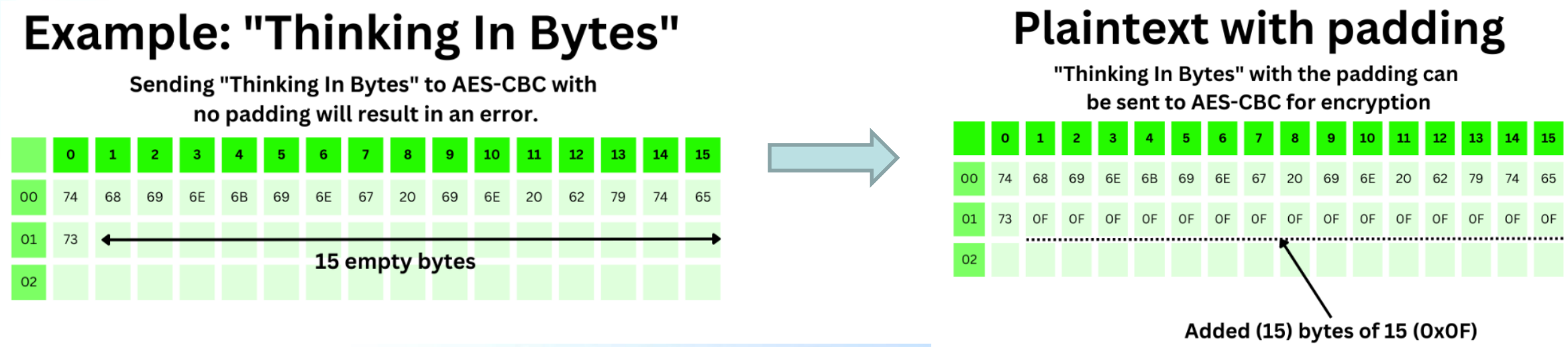


2.4. Principe de Chiffrement avec le mode ECB (Le plus simple)

2.4.1. Etapes de chiffrement AES

1. **Message (texte en clair):** Le message est d'abord divisé en blocs de 128 bits (16 octets)

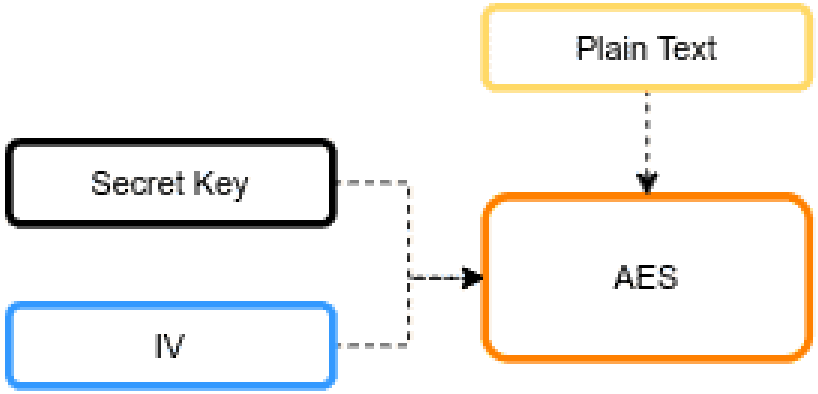
2. **Padding** : Si la taille du message en clair n'est pas un multiple de la taille de bloc d'AES (qui est de 128 bits ou 16 octets), vous devez ajouter un **remplissage (padding)** pour aligner le message sur une taille de bloc multiple.



3. Génération de la Clé (Key):

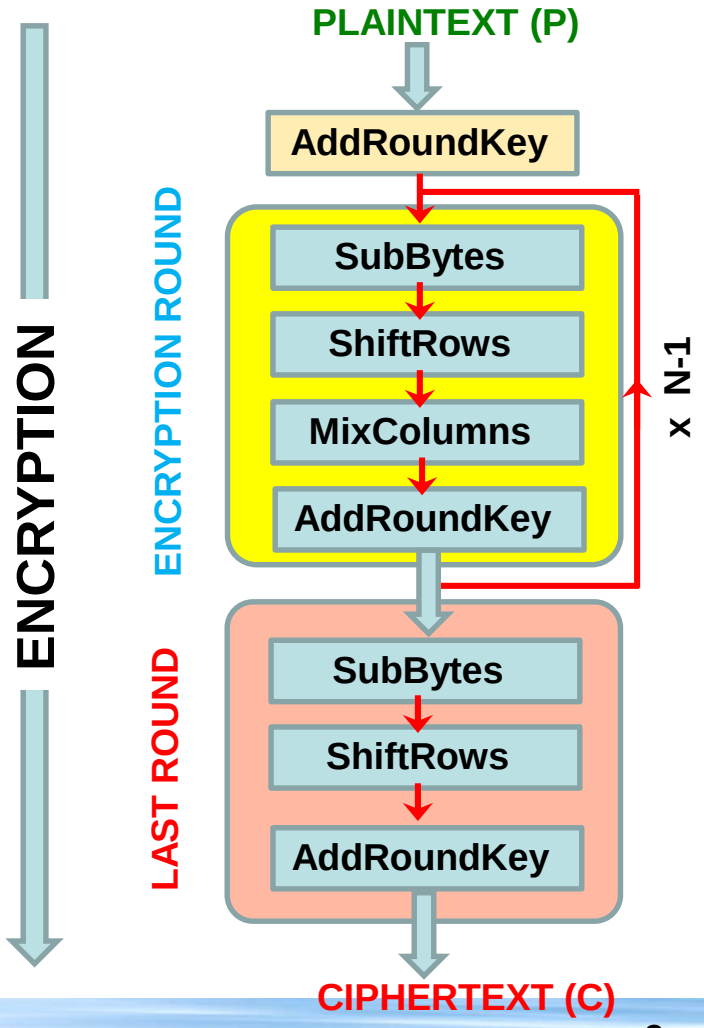
- AES utilise une clé secrète qui doit avoir une longueur de **128**, **192** ou **256** bits.
- Cette clé est utilisée pour réaliser les opérations de chiffrement et de déchiffrement.
- En fonction de la longueur de la clé, AES effectuera un nombre différent de tours (**rounds**) :

  - **128 bits** : **10 tours**
  - **192 bits** : **12 tours**
  - **256 bits** : **14 tours**



4. Chiffrement AES (Etapes pour le cas du mode ECB : plus simple)

- **Chiffrement des blocs de 128 bits** : Pour chaque bloc de données de 128 bits, on applique les **10, 12 ou 14 tours** de chiffrement (selon la taille de la clé) :
- **Tour 1 à N-1** (N dépend de la taille de la clé) :
  - **AddRoundKey** : Le bloc de données (128 bits) est XORé avec la **clé du tour** correspondante.
  - **SubBytes** : Chaque octet du bloc est remplacé par un octet selon la **S-Box**.
  - **ShiftRows** : Les lignes du bloc (tableau 4x4) sont décalées.
  - **MixColumns** : Une multiplication matricielle est effectuée sur les colonnes du bloc pour mélanger les octets (sauf au dernier tour).
- **Dernier tour (Tour N)** :
  - **SubBytes** : Substitution des octets à l'aide de la **S-Box**.
  - **ShiftRows** : Décalage des lignes du bloc.
  - **Pas de MixColumns** : Contrairement aux autres tours, cette étape est omise dans le dernier tour.
  - **AddRoundKey** : Le bloc de données est XORé avec la **clé du dernier tour**.
- **Obtenir le texte chiffré** : Après avoir effectué toutes les étapes de chiffrement sur chaque bloc, le texte chiffré final est constitué de tous les blocs chiffrés combinés. Chaque bloc de 128 bits devient une version chiffrée du texte en clair correspondant.



2.4.2. Matrice d'état 4x4 (state): Bloc AES :

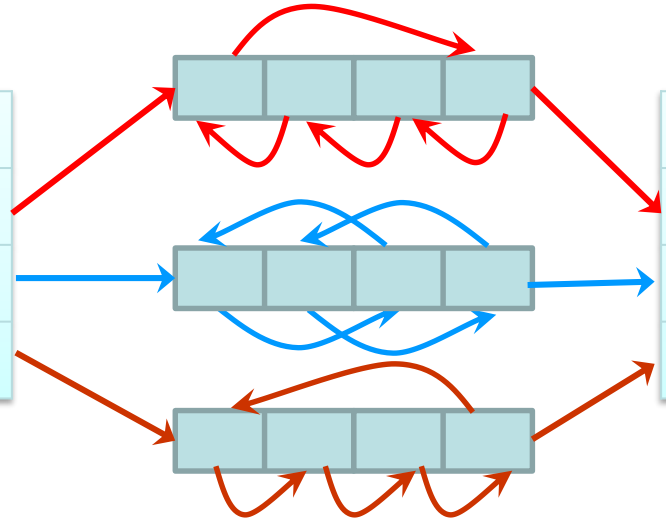
L'état est une matrice de **4x4** utilisée pour stocker les octets du texte en clair ou du texte chiffré (16 octotets), et chaque opération de chiffrement dans AES (comme **ShiftRows**, **SubBytes**, **MixColumns**, et **AddRoundKey**) est effectuée sur cette matrice.

S0	S1	S2	S3
S4	S5	S6	S7
S8	S9	S10	S11
S12	S13	S14	S15

Exemple: **Fonctionnement de ShiftRows**

- **Ligne 1** : Elle reste inchangée.
- **Ligne 2** : Elle est décalée d'un octet vers la gauche.
- **Ligne 3** : Elle est décalée de deux octets vers la gauche.
- **Ligne 4** : Elle est décalée de trois octets vers la gauche.

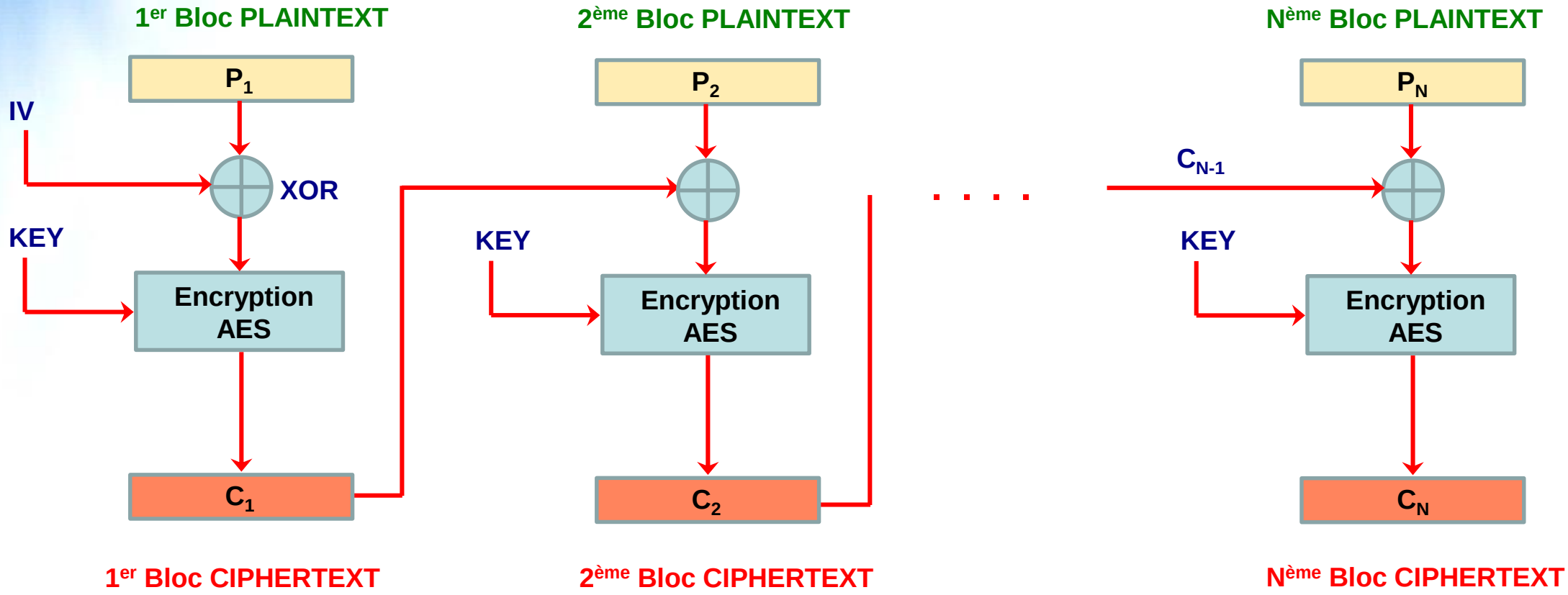
S0	S1	S2	S3
S4	S5	S6	S7
S8	S9	S10	S11
S12	S13	S14	S15



S0	S1	S2	S3
S5	S6	S7	S4
S10	S11	S8	S9
S15	S12	S13	S14

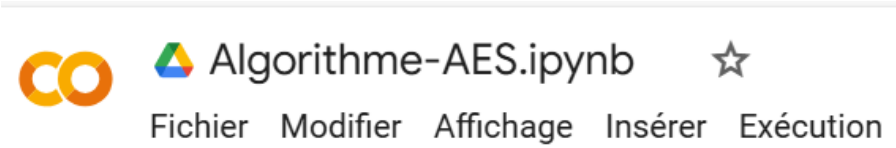
2.5. Principe de Chiffrement avec le mode CBC (Plus sécurisé que ECB)

- **Génération du vecteur d'initialisation (IV):** Un **IV aléatoire** (128 bits ou 16 octets) est généré pour introduire de l'aléa dans le chiffrement (cas des modes par blocs).
- **Initialisation** : Le premier bloc de données est combiné (via un XOR) avec un vecteur d'initialisation (IV) avant d'être chiffré.
- **Enchaînement des blocs** : Pour les blocs suivants, le texte clair de chaque bloc est combiné avec le texte chiffré du bloc précédent (toujours via un XOR) avant d'être chiffré.
- **Finalité** : Cette méthode garantit que les blocs identiques dans le texte clair produisent des blocs chiffrés différents, car ils dépendent des blocs précédents.


1er Bloc PLAINTEXT
2ème Bloc PLAINTEXT
Nème Bloc PLAINTEXT
P1
P2
PN
IV
XOR
CN-1
KEY
KEY
KEY
Encryption AES
Encryption AES
Encryption AES
C1
C2
CN
1er Bloc CIPHERTEXT
2ème Bloc CIPHERTEXT
Nème Bloc CIPHERTEXT

2.6. Exemple d'implémentation de l'AES avec Python :

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
- Renommer le **Algorithme-AES.ipynb**



1. **Avant de pouvoir utiliser AES dans Google Colab, vous devez installer la bibliothèque pycryptodome**

```
!pip install pycryptodome
```

```
Collecting pycryptodome
  Downloading pycryptodome-3.21.0-cp36-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.4 kB)
Downloading pycryptodome-3.21.0-cp36-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 15.2 MB/s eta 0:00:00
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.21.0
```

2. Importer les bibliothèques nécessaires:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
```

3. Génération d'un message à chiffrer:

```
# Message à chiffrer
plaintext = "MASTER ISE DE L'ENSA DE KENITRA : Sécurisé avec AES"
print(f"Message original : {plaintext}")

# Convertir le message en bytes
plaintext_bytes = plaintext.encode()
```

```
Message original : MASTER ISE DE L'ENSA : Sécurisé avec AES
```

4. Génération d'une clé AES et d'un vecteur d'initialisation:

```
# Générer une clé AES de 16 octets (AES-128)
key = get_random_bytes(16)
print(f"Clé AES : {key.hex()}")

# Générer un vecteur d'initialisation (IV) de 16 octets
iv = get_random_bytes(16)
print(f"IV : {iv.hex()}")
```

```
Clé AES : e55d3f327be252502936f2f81cdbc45b
IV : 66465e9b01d6be1a185f8682c978f9d4
```

- *La **clé AES** est utilisée pour **chiffrer** et **déchiffrer** les données avec l'algorithme de chiffrement **AES***
- ***Vecteur d'initialisation (IV)** : Ajoute de l'aléatoire pour garantir que les mêmes données chiffrées avec la même clé ne produisent pas toujours le même texte chiffré. L'IV renforce la sécurité en rendant les chiffrement plus difficile à analyser*

5. Chiffrement AES du message:

```
# Chiffrement AES en mode CBC
cipher = AES.new(key, AES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext_bytes, AES.block_size))  # Ajout de padding
print(f"Texte chiffré (hex) : {ciphertext.hex()}")
```

```
Texte chiffré (hex) : 02d5225e9ec8dfac2386cf8ae8077db68f7841a70f926a7658904b88f6596a59347e9c80d76d40a8083351e9197369ae
```

Chaque caractère hexadécimal (de 0 à f) représente un octet (8 bits) du texte chiffré.

6. Déchiffrement AES:

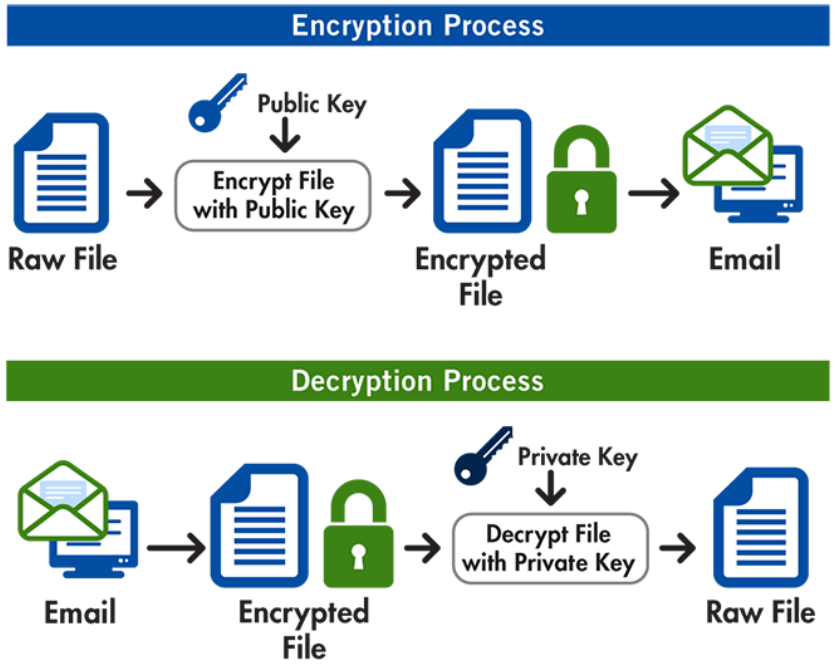
```
# Déchiffrement AES en mode CBC
decipher = AES.new(key, AES.MODE_CBC, iv)
decrypted_padded_bytes = decipher.decrypt(ciphertext)
decrypted_bytes = unpad(decrypted_padded_bytes, AES.block_size)  # Suppression du padding
decrypted_text = decrypted_bytes.decode()
print(f"Message déchiffré : {decrypted_text}")
```

```
Message déchiffré : MASTER ISE DE L'ENSA : Sécurisé avec AES
```

3. Chiffrement Asymétrique

3.1. Clé Publique / Clé Privée:

- Dans un système de **chiffrement asymétrique** (comme RSA ou ECC), il y a deux **clés distinctes** :
  - **Clé publique** : elle est utilisée pour **chiffrer** le message. Elle peut être partagée librement avec tout le monde.
  - **Clé privée** : elle est utilisée pour **déchiffrer** le message. Elle doit être gardée secrète et uniquement accessible à la personne qui a la clé privée correspondante.
- La **signature numérique** est un procédé cryptographique qui permet de garantir l'intégrité, l'authenticité et l'origine d'un message en l'associant à une **clé privée**, permettant à toute personne disposant de la **clé publique** correspondante de vérifier que le message n'a pas été modifié et qu'il provient bien de l'expéditeur.



3.2. RSA (Rivest-Shamir-Adleman)

Fonctionnement :

- RSA est un algorithme de cryptographie asymétrique basé sur l'utilisation d'une paire de clés : une clé publique et une clé privée. Voici les étapes principales de son fonctionnement :

1. Génération des clés :

- Deux grands nombres premiers, souvent notés p et q, sont choisis.
- Le produit $n = p \times q$ sert de base pour les clés publiques et privées.
- On calcule $\phi(n)$ appelé **indicatrice d'Euler**, qui joue un rôle dans la construction des clés:

$$\phi(n) = (p-1)(q-1)$$

- Une clé **publique** e est choisie, avec deux conditions :

  a) e est un entier tel que $1 < e < \phi(n)$
  b) e est **premier avec** $\phi(n)$ (c'est-à-dire que $\mathbf{PGCD}\left(e, \phi(n)\right) = 1$

- La clé **privée** d est l'**inverse modulaire** de e $modulo$ $\phi(n)$. Cela signifie que d est un nombre entier qui vérifie la relation suivante :

$$(e \times d) \bmod \phi(n) = 1$$

Autrement dit, lorsque $e \times d$ est divisé par $\phi(n)$, le reste est 1.

L'inverse modulaire d est calculé en utilisant **l'algorithme d'Euclide étendu**.

Exemple numérique:

Supposons :

- $p = 7$, $q = 11$.
- $n = p \times q = 77$.
- $\phi(n) = (p-1)(q-1) = 60$.
- Choisissons $e = 7$, qui est premier avec $\phi(n) = 60$.
- Trouvons d tel que $(e \times d) \bmod \phi(n) = 1$, soit :

$$(7 \times d) \bmod 60 = 1$$

- En utilisant l'algorithme d'Euclide étendu (voir code Python ci-contre), on trouve : $d = 43$

- Ainsi, $(7 \times 43) \bmod 60 = 301 \bmod 60 = 1$.
- Donc, la clé privée est $d = 43$.

```python
from sympy import mod_inverse

# Données
e = 7
phi_n = 60

# Calcul de l'inverse modulaire
d = mod_inverse(e, phi_n)
d
```

```
43
```

*Deux nombres entiers sont dits **premiers entre eux** lorsqu'ils n'ont aucun diviseur commun autre que 1*

2. Cryptage avec la clé publique :

- Lorsqu'un message M doit être chiffré, il est transformé en un nombre C grâce à la formule :

$$C = M^e \mod n$$

Message est d'abord **converti en un nombre (code ASCII) avant d'appliquer l'exposant**

- Ce chiffre C peut ensuite être envoyé de manière sécurisée.

3. Décryptage avec la clé privée :

- Le destinataire utilise la clé privée d pour retrouver le message original :

$$M = C^d \mod n$$

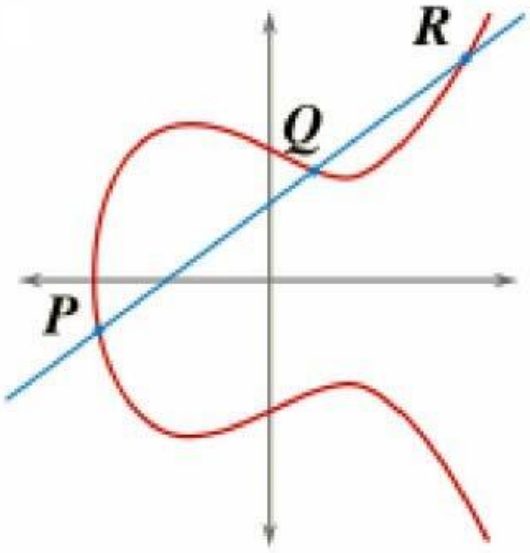
L'avantage principal de RSA est que la clé publique peut être partagée librement pour le chiffrement, tandis que la clé privée, gardée secrète, est utilisée pour le décryptage.

3.3. ECC (Elliptic Curve Cryptography)

- **ECC**: Cryptographie à Courbes Elliptiques.
- Les courbes **Elliptiques** sont des courbes définies par une équation mathématique du type:

$$y^2 = x^3 + ax + b$$

a et b sont deux constantes



$$y^2 = x^3 + ax + b$$

- **Opérations de base** : Les points sur la courbe peuvent être additionnés (addition de points) ou multipliés par un scalaire. Ces opérations sont difficiles à inverser, ce qui les rend sécurisées pour la cryptographie

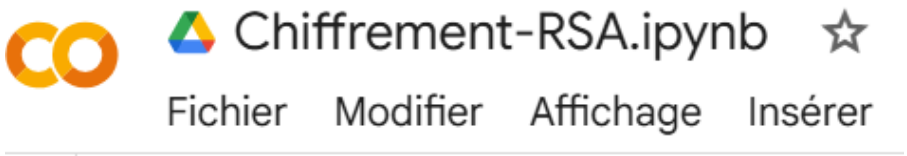
- ***Addition de points*** *: Deux points P et Q sur une courbe elliptique peuvent être additionnés pour donner un troisième point R.*
- ***Multiplication scalaire*** *: La multiplication d'un point P par un entier k consiste à additionner plusieurs copies de PPP. Cela est l'opération clé utilisée dans ECC, où la difficulté de cette multiplication fait la sécurité de l'algorithme.*

3.4. RSA vs. ECC

Critère	RSA	ECC
Taille des clés	Clés très grandes (2048 à 4096 bits pour une bonne sécurité).	Clés beaucoup plus petites (256 bits offrent une sécurité équivalente à RSA 3072 bits).
Efficacité	Plus lent à cause des calculs sur de grandes clés.	Plus rapide grâce à des calculs sur des clés plus petites.
Consommation	Nécessite plus de mémoire, de puissance et de bande passante.	Moins gourmande en ressources (idéal pour IoT et appareils à faible puissance).
Sécurité	Basé sur la difficulté de factoriser de grands nombres.	Basé sur le problème du logarithme discret sur une courbe elliptique.
Applications typiques	Utilisé pour le chiffrement (TLS, signatures, etc.).	Utilisé pour le chiffrement et les signatures numériques, surtout dans des environnements contraints.
Avantage principal	Bien établi, robuste, et largement utilisé.	Sécurité équivalente avec moins de ressources. Parfait pour les appareils modernes.

3.5. Exemple d'implémentation de RSA avec Python: Utilisation de la bibliothèque PyCryptodom

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
- Renommer le **Chiffrement-RSA.ipynb**



1. Installer la bibliothèque pycryptodome

```
# Installer PyCryptodome
!pip install pycryptodome
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64
```

```
Collecting pycryptodome
  Downloading pycryptodome-3.21.0-cp36-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.4 kB)
Downloading pycryptodome-3.21.0-cp36-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (2.3 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.3/2.3 MB 21.4 MB/s eta 0:00:00
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.21.0
```

2. Générer les clés RSA (Privée et Publique)

Génération des éléments RSA:

- **Clé publique** : modulus ***n***, exposant ***e***
- **Clé privée** : n, d, p, q, d_p, d_q, q_{inv}

```
# 1. Génération des clés RSA
key = RSA.generate(2048)  # Génération d'une clé RSA de 2048 bits
private_key = key.export_key()  # Clé privée
public_key = key.publickey().export_key()  # Clé publique

print("Clé privée:", private_key.decode())
print("\nClé publique:", public_key.decode())
```

```
Clé privée: -----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA2Laay+hfqRyGFpjYWQ1piYZ5P8c2FMmoS4JoymPppIvCX3wp
ue8+JKX6rVKhGKn84Qmce0C1vT+RyHx8hyyiWNp5nprF23OjF1XPI8EA91cVlyQw
H3Eqj48PGeiOm8lTZDz17J9v5X+BdDVbtSAxuvGDAYb+GL75Sil86vap49xdnkMe
```

```
Clé publique: -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2Laay+hfqRyGFpjYWQ1p
iYZ5P8c2FMmoS4JoymPppIvCX3wpue8+JKX6rVKhGKn84Qmce0C1vT+RyHx8hyyi
WNp5nprF23OjF1XPI8EA91cVlyQwH3Eqj48PGeiOm8lTZDz17J9v5X+BdDVbtSAx
```

3. Définir la fonction de Chiffrement:

```
# 2. Chiffrement
def encrypt_message(message, public_key):
    rsa_public_key = RSA.import_key(public_key)
    cipher = PKCS1_OAEP.new(rsa_public_key)
    encrypted_message = cipher.encrypt(message.encode())  # Chiffrement
    return base64.b64encode(encrypted_message)  # Encodage en Base64 pour lisibilité
```

4. Définir la fonction de Déchiffrement:

```
# 3. Déchiffrement
def decrypt_message(encrypted_message, private_key):
    rsa_private_key = RSA.import_key(private_key)
    cipher = PKCS1_OAEP.new(rsa_private_key)
    decrypted_message = cipher.decrypt(base64.b64decode(encrypted_message))  # Décodage
    return decrypted_message.decode()
```

5. Test de l'algorithme RSA:

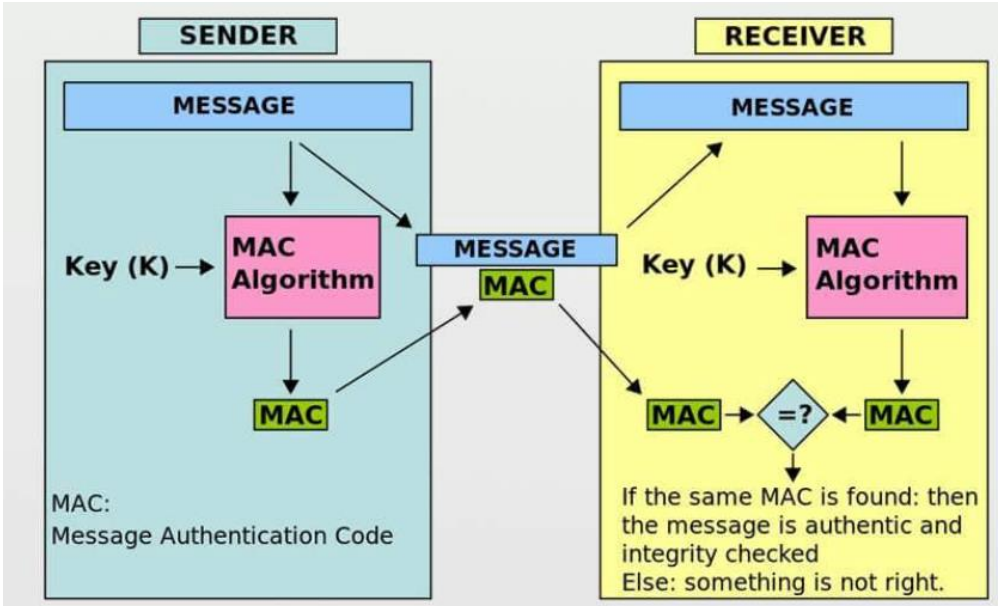
```
# Test complet
message = "ISE: MASTER INGENIRIE DES SYSTEMES EMBARQES"
print("\nMessage original:", message)

# Chiffrement
cipher_text = encrypt_message(message, public_key)
print("\nMessage chiffré:", cipher_text)

# Déchiffrement
decrypted_message = decrypt_message(cipher_text, private_key)
print("\nMessage déchiffré:", decrypted_message)
```

```
Message original: ISE: MASTER INGENIRIE DES SYSTEMES EMBARQES

Message chiffré: b'pjKXlw3edh82zddoDa1eIp4mPvXDTP7HPW9e0gOwPKwVOh(

Message déchiffré: ISE: MASTER INGENIRIE DES SYSTEMES EMBARQES
```

4. Authentification et Intégrité

4.1. Définition

- **HMAC (Hashed Message Authentication Code): "Authentification et intégrité :**
- HMAC est un mécanisme de cryptographie symétrique utilisé pour garantir à la fois l'intégrité et l'authenticité d'un message.
  - **Authenticité** : Assurer que le message provient de l'expéditeur légitime.
  - **Intégrité** : Vérifier que le message n'a pas été modifié pendant la transmission.
- Couramment utilisé dans les protocoles de communication sécurisés tels que HTTPS, TLS, et les API REST



4.2. Fonction HMAC:

- La fonction HMAC est définie comme suit:

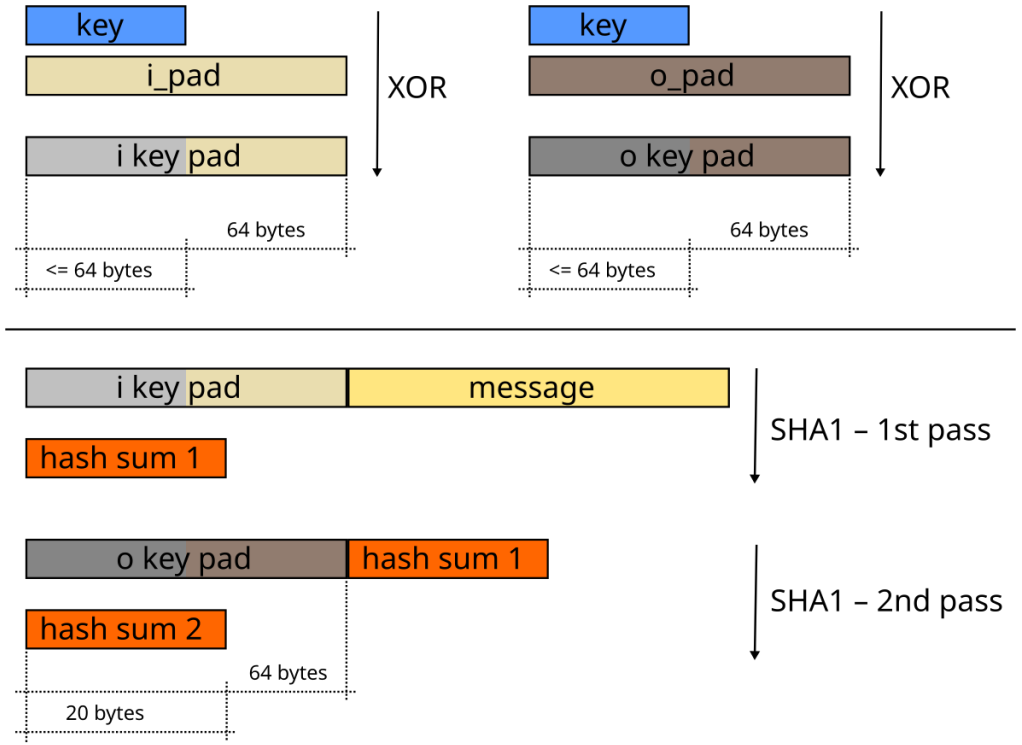
$$HMAC_K(m) = h\Big((K \oplus opad) \parallel h\big((K \oplus ipad) \parallel m\big)\Big)$$

avec :

- ***h*** : une fonction de hachage itérative,
- ***K*** : la clé secrète, hachée par la fonction h si plus longue que sa taille de bloc, puis complétée avec des zéros pour qu'elle atteigne la taille de bloc de la fonction h
- ***m*** : le message à authentifier,
- "**||**" désigne une **concaténation** et " ⊕ " un « ou » exclusif,
- ***ipad*** et ***opad***, chacune de la taille d'un bloc, sont définies par : *ipad* = 0x363636...3636 et *opad* = 0x5c5c5c...5c5c. Donc, si la taille de bloc de la fonction de hachage est 512 bits, *ipad* et *opad* sont 64 répétitions des octets, respectivement, 0x36 et 0x5c.

4.3. Fonctionnement de HMAC

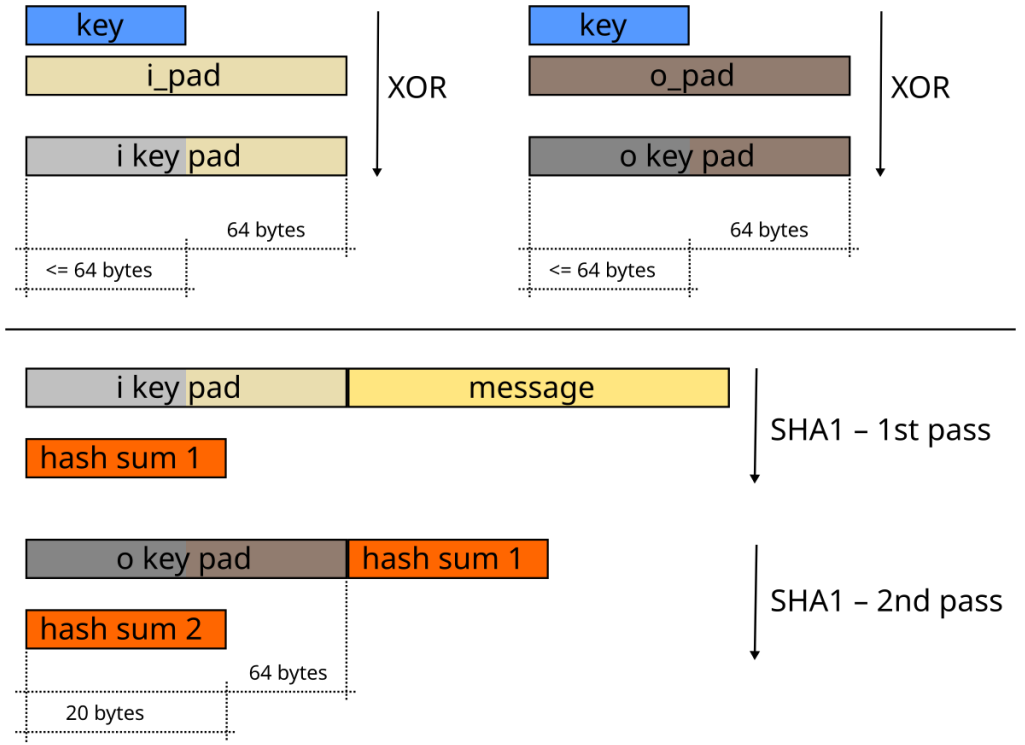
1. **Entrées** :
   - **Message** : Le message à authentifier (peut être de n'importe quelle taille).
   - **Clé secrète partagée** : La clé utilisée pour créer l'HMAC. Elle doit être partagée entre l'émetteur et le récepteur.
   - **Fonction de hachage** : La fonction de hachage utilisée, comme SHA-1, SHA-256, etc.



2. **Création de deux valeurs dérivées de la clé** :
   - **Key XOR opad (clé XOR avec le padding)** : L'**opad** (Outer Padding) est un vecteur constant (souvent 0x5c) qui est XORé avec la clé.
   - **Key XOR ipad (clé XOR avec le padding interne)** : L'**ipad** (Inner Padding) est également un vecteur constant (souvent 0x36) qui est XORé avec la clé.

3. Calcul de l'HMAC :

- Le message est d'abord **concaténé** avec la clé XORée avec **ipad**, et le résultat est haché à l'aide de la fonction de hachage (par exemple SHA-256).
- Ensuite, le résultat obtenu est **concaténé** avec la clé XORée avec opad, et cette concaténation est hachée à nouveau.



4. Résultat :Le résultat final de l'HMAC est le hachage produit par cette double opération, qui est une **signature** du message.

4.4. Exemple d'implémentation de processus de HMAC avec Python:

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
- Renommer le **HMAC.ipynb**

1. Bibliothèques, message à chiffré et clé

```
# 1. Importation des bibliothèques nécessaires
import hmac
import hashlib
import binascii

# 2. Définition du message et de la clé
message = b"MASTER ISE CHIFFRE AVEC HMAC"
key = b"MUS-ISE"

# 3. Taille des blocs de la fonction de hachage (pour SHA-256, c'est 64 octets)
block_size = 64
```

b: chaque caractère de la chaîne est stocké sous forme d'octet (byte) dans la mémoire

2. Création des Fonctions de pads

```python
# 4. Fonction pour créer les pads internes et externes
def create_pads(key, block_size):
    # Si la clé est plus grande que la taille du bloc, on la hache
    if len(key) > block_size:
        key = hashlib.sha256(key).digest()

    # Si la clé est plus petite que la taille du bloc, on la complète avec des zéros
    if len(key) < block_size:
        key = key + b'\x00' * (block_size - len(key))

    # Création du pad interne et externe
    ipad = bytes([x ^ 0x36 for x in key])
    opad = bytes([x ^ 0x5C for x in key])

    return ipad, opad
```

Chaque élément **x** de la séquence key est **XORé** (^) avec la valeur en hexadécimal **0x36**

3. Création des pads et leur affichage

```
# 5. Création des pads internes et externes
ipad, opad = create_pads(key, block_size)

# 6. Affichage des pads
print(f"Pad interne (ipad) : {binascii.hexlify(ipad)}")
print(f"Pad externe (opad) : {binascii.hexlify(opad)}")
```

```
Pad interne (ipad) : b'7b63651b7f6573636363636363636363636363636363636363636363636363
Pad externe (opad) : b'11090f71150f195c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c
```

4. HMAC par étapes

```
# 7. HMAC étape par étape

# a. Application du pad interne + message
inner = hashlib.sha256(ipad + message).digest()
print(f"Hash intermédiaire (ipad + message) : {binascii.hexlify(inner)}")

# b. Application du pad externe + résultat de la première étape
hmac_result = hashlib.sha256(opad + inner).digest()
print(f"HMAC final : {binascii.hexlify(hmac_result)}")
```

Résultat HMAC

```
Hash intermédiaire (ipad + message) : b'e0185f15d58c5dcaeb29bd41ea715ded29cf7fb2fffa8f2128f2125b9f21024e'
HMAC final : b'74f1d3da149a7a1d0dd5b2aaf463f697eb903d82ea9fe5146cfd111c667281ce'
```

5. Affichage de HMAC

```
# 8. Affichage du HMAC en hexadécimal
print(f"HMAC (en hexadécimal) : {binascii.hexlify(hmac_result).decode()}")
```

```
HMAC (en hexadécimal) : 74f1d3da149a7a1d0dd5b2aaf463f697eb903d82ea9fe5146cfd111c667281ce
```

5. Vérification de HMAC

À ce point, vous pouvez vérifier le HMAC en le recalculant et en le comparant avec un HMAC reçu

```
# Simulons que vous avez un HMAC reçu pour le message
received_hmac = hmac_result  # Cela correspond au HMAC qu'on a généré ci-dessus

# Vérification
def verify_hmac(received_hmac, key, message):
    calculated_hmac = hmac.new(key, message, hashlib.sha256).digest()
    print("HMAC reçu      :", received_hmac)
    print("HMAC calculé   :", calculated_hmac)
    if calculated_hmac == received_hmac:
        print("Le HMAC est valide et le message est authentique.")
    else:
        print("Le HMAC est invalide ou le message a été modifié.")
verify_hmac(received_hmac, key, message)
```

```
HMAC reçu      : b't\xf1\xd3\xda\x14\x9az\x1d\r\xd5\xb2\xaa\xf4c\xf6\x97\xeb\x90=\x82\xea\x9f\xe5\x14l\xfd\x11\x1cfr\x81\xce'
HMAC calculé   : b't\xf1\xd3\xda\x14\x9az\x1d\r\xd5\xb2\xaa\xf4c\xf6\x97\xeb\x90=\x82\xea\x9f\xe5\x14l\xfd\x11\x1cfr\x81\xce'
Le HMAC est valide et le message est authentique.
```

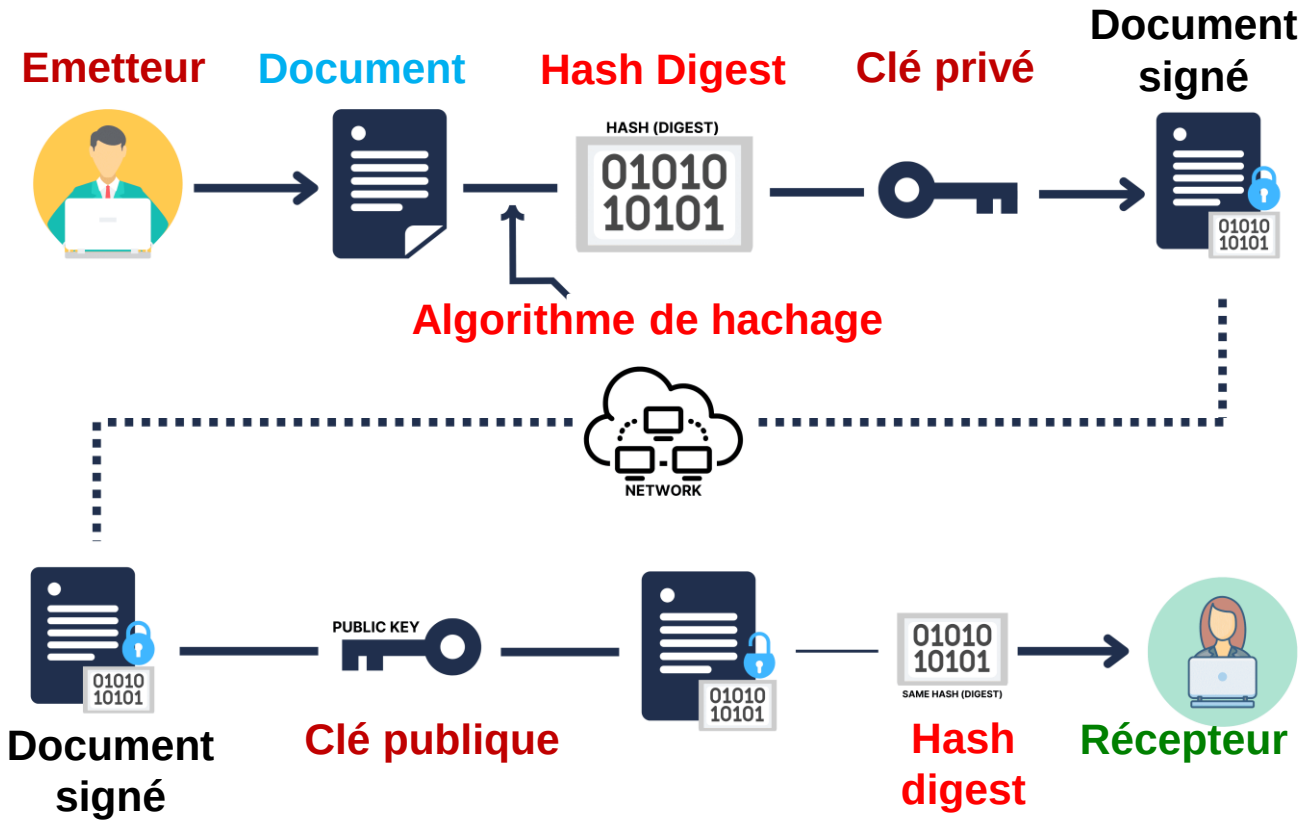
5. Signature Numérique

5.1. Objectifs et Rôles:

- La signature numérique est une technique cryptographique qui permet d'assurer l'intégrité, l'authenticité et la non-répudiation des messages ou documents électroniques.
- Elle repose sur des mécanismes de cryptographie asymétrique.
- Elle garantit trois propriétés essentielles :
  - **Authenticité** : Assure que l'expéditeur du message est bien celui qu'il prétend être.
  - **Intégrité** : Garantit que le message n'a pas été modifié après sa signature.
  - **Non-répudiation** : L'auteur ne peut nier avoir signé le message.

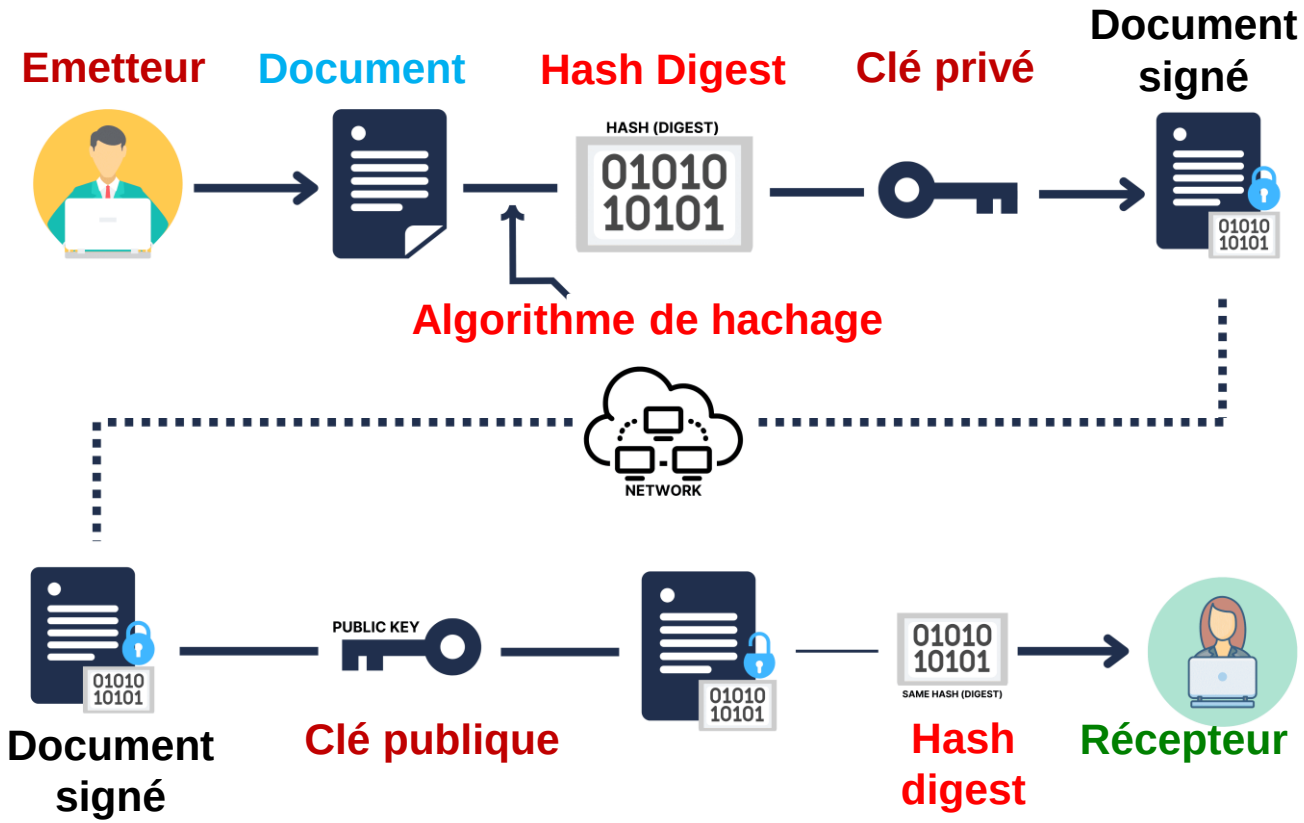
5.2. Processus de Signature :

1. Le document est soumis à une fonction de hachage (ex: **SHA-256**) pour obtenir un **condensat** (Hash digest) unique.
2. Ce condensat est ensuite chiffré avec la **clé privée** du signataire, ce qui constitue la **signature numérique (**algorithmes: RSA, DSA (Digital Signature Algorithm), Ed25519, …)
3. Le document et la signature sont envoyés au destinataire.



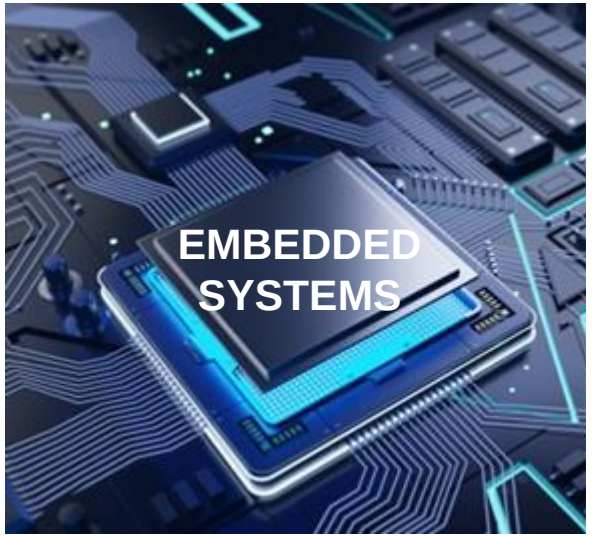
5.3. Processus de Vérification :

1. Le destinataire applique la même fonction de hachage sur le document reçu pour générer un nouveau condensat.
2. Il déchiffre la signature reçue avec la **clé publique** de l'expéditeur.
3. Il compare le condensat (Hash digest) déchiffré avec celui calculé :
   a) S'ils correspondent, la signature est valide.
   b) Sinon, le document a été modifié ou l'expéditeur n'est pas authentique.



5.4. Applications de la Signature Numérique:

- **Certificats SSL/TLS** : Authentification des sites web.
- **Blockchain** : Validation des transactions (ex: Bitcoin, Ethereum (contrats intelligents)).
- **Documents officiels** : Contrats électroniques, e-mails signés.
- **Authentification des logiciels** : Signature des mises à jour et des programmes.



FIN !
Questions ?