AES: ADVANCED ENCRYPTION STANDARD

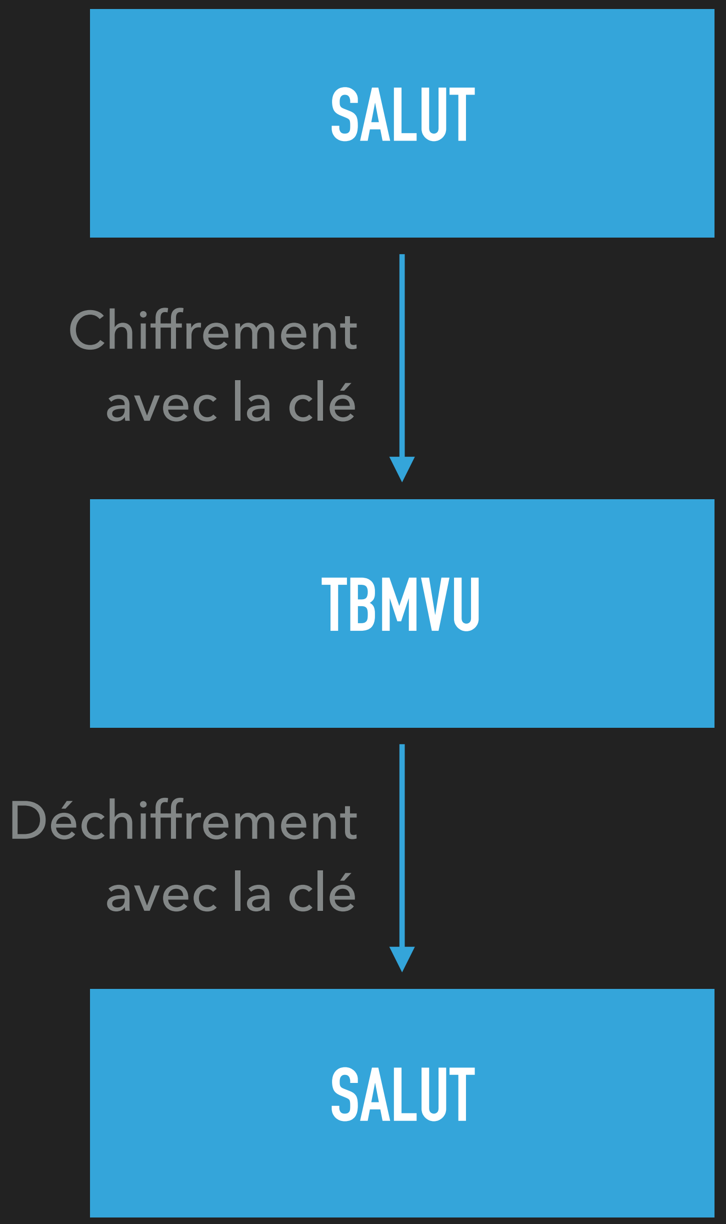
CRYPTOGRAPHIE SYMÉTRIQUE

PLAN DE NOTRE PRÉSENTATION D'AUJOURD'HUI

- La cryptographie, c'est quoi ?
- Introduction aux corps finis
- Structure de l'algorithme AES
- Renforcement à l'aide des ciphers

CHIFFREMENT DE DONNÉES

- Echange de données de manière sécurisée
- Utilisation d’une clé (symétrique ou asymétrique)
- Différents standards : DES, AES, RSA, …



LE CORPS $GF(256)$ (OU $GF(2^8)$)

- Ensemble à 256 éléments
- Polynômes de degré inférieur ou égal à 7, avec coefficients 0 ou 1 (exemple: $X^4 + X^2 + 1 \in GF(2^8)$)
- $+$ et $\times$ sont des lois de composition interne :
  $\forall P, Q \in GF(2^8), P + Q \in GF(2^8), P \times Q \in GF(2^8)$

POURQUOI CHOISIR $GF(2^8)$?

- Données binaires : 1 octet = 256 valeurs possibles
- Bijection entre des données binaires et une suite d'éléments de $GF(2^8)$
- Exemple : A ⇔ $(1000001)_2 \Leftrightarrow X^7 + 1 \in GF(2^8)$

REPRÉSENTATION DES DONNÉES

- Utilisation d'une matrice $M \in M_4(GF(2^8))$ à 16 coefficients
- Représente un bloc de 16 octets

RÉPARTITION EN ÉTAPES

- Une matrice en entrée et en sortie
- 4 étapes : substitution, décalage, mixage et ajout de la clé
- Répétition de ces étapes entre 10 et 14 fois

```
let rec tour entree clefs n =
    (* On récupère la clé du tour *)
    let cle = clefs.(11-n) in

    match n with
    (* Dernier tour, sans le mixage *)
    | 1 -> ajout (decalage (substitution entree)) cle

    (* Tour normal, qu'on envoie au tour suivant *)
    | _ -> tour (ajout (mixage (decalage (substitution entree)) false) cle) clefs (n-1)
```

SUBSTITUTION

- Bijection $S : GF(2^8) \to GF(2^8)$
- Permet la non linéarité de l'opération

```
let sbox = [|
    0x63; 0x7c; 0x77; 0x7b; 0xf2; 0x6b; 0x6f; 0xc5; 0x30; 0x01; 0x67; 0x2b; 0xfe; 0xd7; 0xab; 0x76;
    …
    0x8c; 0xa1; 0x89; 0x0d; 0xbf; 0xe6; 0x42; 0x68; 0x41; 0x99; 0x2d; 0x0f; 0xb0; 0x54; 0xbb; 0x16
|]

let substitution entree = Array.map (fun x -> sbox.(x)) entree
```

DÉCALAGE

- Permutation de coefficients
- Evite que les colonnes soient chiffrées séparément

```
let decalage entree =
    [|
        (* 0 <- *)      (* 1 <- *)      (* 2 <- *)      (* 3 <- *)
        entree.(0);     entree.(5);     entree.(10);    entree.(15);
        entree.(4);     entree.(9);     entree.(14);    entree.(3);
        entree.(8);     entree.(13);    entree.(2);     entree.(7);
        entree.(12);    entree.(1);     entree.(6);     entree.(11)
    |]
```

MIXAGE

- Produit entre les colonnes :

$$M : \begin{bmatrix} a_i \\ a_{i+1} \\ a_{i+2} \\ a_{i+3} \end{bmatrix} \mapsto \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_i \\ a_{i+1} \\ a_{i+2} \\ a_{i+3} \end{bmatrix}$$

- Evite que les lignes soient chiffrées séparément

```
let mixage entree inverse =
    let sortie = Array.make 16 0 in
    for i = 0 to 3 do
        (* On extrait la colonne *)
        let colonne = [| entree.(i*4); entree.(i*4 + 1); entree.(i*4 + 2); entree.(i*4 + 3) |] in

        (* On fait le produit et on place les coefficients *)
        let nouvelle_colonne = produit_colonne colonne inverse in
        sortie.(i*4) <- nouvelle_colonne.(0);
        sortie.(i*4 + 1) <- nouvelle_colonne.(1);
        sortie.(i*4 + 2) <- nouvelle_colonne.(2);
        sortie.(i*4 + 3) <- nouvelle_colonne.(3)
    done;
    sortie
```

STRUCTURE DE L'ALGORITHME AES

```
let produit_colonne col inverse =
    (* On fabrique la colonne de sortie *)
    let resultat = Array.make 4 0 in
    let used = if inverse then rm else m in
    for i = 0 to 3 do
        (*
        * Chaque coefficient est la somme des
        * produits des éléments d'une ligne
        * avec ceux d'une colonne
        *)
        for k = 0 to 3 do
            resultat.(i) <- (used.(i*4 + k) ** col.(k)) lxor resultat.(i)
        done
    done;
    resultat
```

```
let rec reste dividende diviseur =
    (* Degrés des polynômes *)
    let d1 = degre dividende in
    let d2 = degre diviseur in
    (* On regarde lequel a le plus grand degré *)
    if d1 >= d2 then
        (*
        * Si c'est le dividende, on multiplie le diviseur par x à la
        * puissance la différence des dégrées, et on soustrait ce résultat
        * au dividende. Le reste de p1 par p2 est donc récursivement le reste
        * de la disivion de ce nouveau polynôme par p2.
        *)
        let quotient = polynome (1 lsl (d1-d2)) in
        let cequonsoustrait = produit diviseur quotient in
        let cequonredivise = somme dividende cequonsoustrait in
        reste cequonredivise diviseur
    else
        (* Sinon on ne peut pas diviser et alors le dividende est le reste *)
        dividende
```

```
let irreductible = [1; 1; 0; 1; 1; 0; 0; 0; 1]

let ( ** ) a b =
    let p1 = polynome a in
    let p2 = polynome b in
    let p = produit p1 p2 in
    let r = reste p irreductible in
    nombre r
```

AJOUT DE LA CLÉ

- Somme de l'entrée avec la clé
- Rend le chiffrement unique à chaque clé

```
let ajout entree cle = Array.map2 (lxor) entree cle
```

QU'EN EST T'IL DU DÉCHIFFREMENT ?

- Bijection réciproque :

$$(A \circ M \circ D \circ S)^{-1} = S^{-1} \circ D^{-1} \circ M^{-1} \circ A^{-1}$$

```
let rec tour_inverse entree clefs n =
    (* On récupère la clé du tour *)
    let cle = clefs.(n) in

    match n with
    (* Premier tour, sans le mixage *)
    | 10 -> tour_inverse (substitution_inverse (decalage_inverse (ajout entree cle))) clefs (n-1)

    (* Dernier tour *)
    | 1 -> substitution_inverse (decalage_inverse (mixage (ajout entree cle) true))

    (* Tour normal, qu'on envoie au tour suivant *)
    | _ -> tour_inverse (substitution_inverse (decalage_inverse (mixage (ajout entree cle) true))) clefs (n-1)
```

PROBLÈME DE L'ALGORITHME

- Algorithme non linéaire :

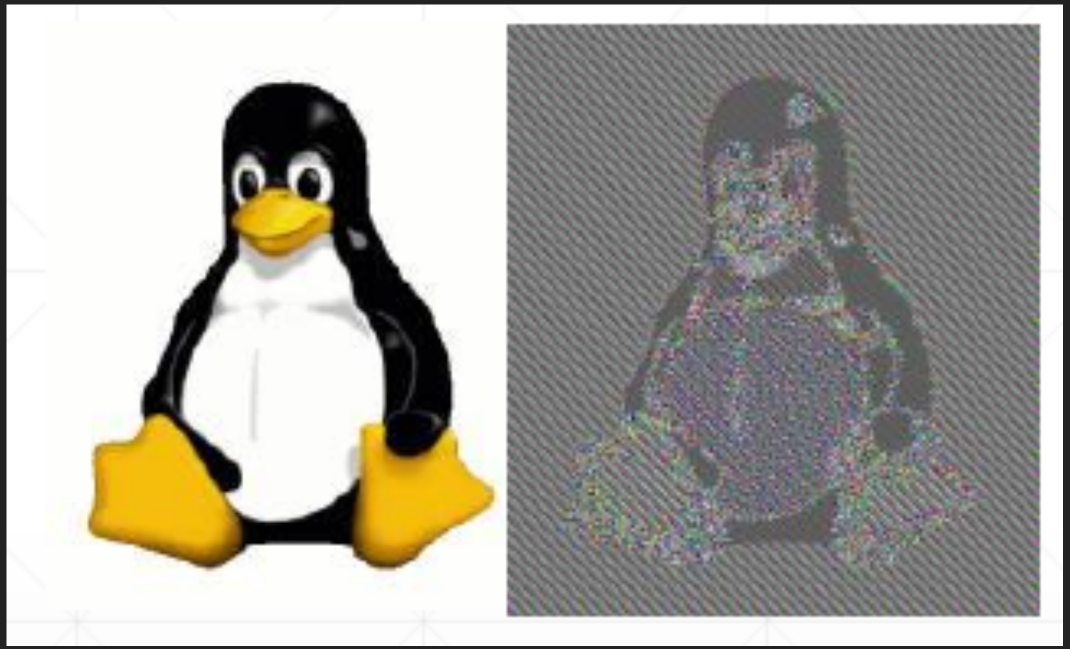
Avec la clé 2B7E151628AED2A6ABF7158809CF4F3C :

000102030405060708090A0B0C0D0E0F -> 50FE67CC996D32B6DA0937E99BAFEC60

010102030405060708090A0B0C0D0E0F -> 38C20C1333E8B7EB738F09DDE66C62AB

- Mais…

Un même bloc sera toujours chiffré de la même manière avec la même clé



LES CIPHERS À LA RESCOUSSE !

- Ajout d'un vecteur d'initialisation à chaque chiffrement
- Résultat dépendant de la clé, mais aussi du bloc précédent

Avec la clé 2B7E151628AED2A6ABF7158809CF4F3C et le cipher CBC :

000102030405060708090A0B0C0D0E0F -> 50FE67CC996D32B6DA0937E99BAFEC60

000102030405060708090A0B0C0D0E0F -> 63A04FC0E2424B29518DCED16F97D529

```
class cbc cle vi =
  object (self)
    inherit cipher cle
    val vi = vi

    method encrypt entree =
      let xored = Array.map2 (lxor) entree vi in
      chiffrer xored cle

    method decrypt entree =
      let decrypted = dechiffrer entree cle in
      Array.map2 (lxor) decrypted vi
  end
```

AES + CIPHERS = 🔒

- Algorithme de chiffrement symétrique non linéaire sécurisé
- Utilisé partout (web, disques, …)
- Pour les curieux : https://github.com/NAS-TIPE/TIPE (documents plus détaillés et code source complet)