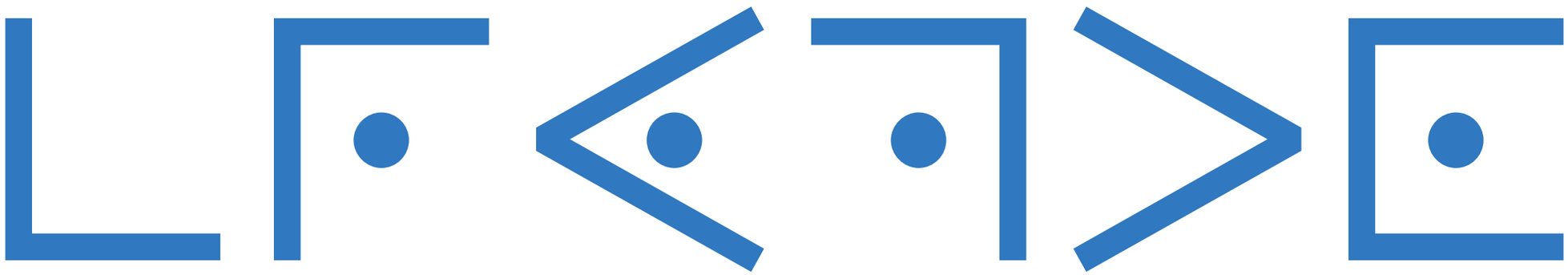


chiffrement symétrique

Nicolas Ollinger

M1 informatique — 2025/2026

Dans les épisodes précédents

Cryptographie à clé secrète

Un schéma de chiffrement à clé secrète est défini par un espace des messages et 3 algorithmes :

$\mathscr{M}$ espace des messages

$\mathrm{Gen}() = k$ algo probabiliste de génération de clé

$\mathrm{Enc}_k(m) = c$ algo probabiliste de chiffrement

$\mathrm{Dec}_k(c) = m$ algo déterministe de déchiffrement

$$\forall k = Gen() \quad \forall m \in \mathscr{M} \quad \mathrm{Dec}_k(\mathrm{Enc}_k(m)) = m$$

Secret parfait (formel)

Le schéma de chiffrement (Gen, Enc, Dec) **assure le secret parfait** si quelque soit la distribution de probabilité sur l'espace des messages, quelque soit le message m et quelque soit le message chiffré c de probabilité non nulle, on a

$$P(M = m \mid C = c) = P(M = m)$$

Formulation « expérimentale »

Un adversaire A choisit deux messages m_0 et m_1 quelconques.

L'un de ces deux messages est choisi uniformément et chiffré en c.

A doit alors deviner si le message chiffré c correspond à m_0 ou à m_1.

Un schéma est **parfaitement indistinguable** si A ne peut pas deviner la bonne réponse avec un probabilité supérieure à 1/2.

Équivalence et limitation

Un schéma de chiffrement assure le secret parfait si et seulement si il est parfaitement indistinguable.

Théorème Si (Gen, Enc, Dec) assure le secret parfait alors l'espace des clés est au moins aussi grand que l'espace des messages.

Sécurité calculatoire

Remplacer le secret parfait par un secret relatif à la **puissance de calcul de l'attaquant**.

Une méthode de chiffrement est sûre si le meilleur algorithme pour casser le chiffre nécessite un nombre d'opérations trop grand pour être utilisable en pratique.

Sécurité sémantique

Sécurité concrète

Un schéma est **(t,ε)-sécurisé** si un adversaire disposant d'un temps de calcul au plus t peut casser le chiffrement avec probabilité au plus ε.

Exemple Force brute en temps t pour un clé de taille n qui réussit avec probabilité au plus

$$ct/2^n$$

Quelques grandeurs

Un processeur à 4GHz exécute 2^{60} cycles en

$$2^{60}/(4 \times 10^9) \text{ s} \approx 9 \text{ ans}$$

Le supercalculateur le plus puissant en juin 2018 développe **122,3 PetaFLOPS**. Il exécute 2^{60} instructions en virgule flottante en

$$2^{60}/(122{,}3 \times 10^{15}) \text{ s} \approx 9{,}427 \text{ s}$$

Pour une attaque force brute sur une clé de **128 bits**, il lui faut 2^{68} fois plus de temps soit environ 6400 fois l'âge estimé de l'univers.

Sécurité asymptotique

Paramétrer les schémas cryptographiques, ainsi que les acteurs (alliés et adversaires) par un **paramètre de sécurité** n.

Les schémas sont initalisés pour une valeur fixée de n (la longueur de la clé).

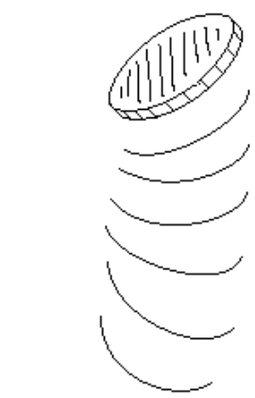
On demande aux **algorithmes efficaces** de s'exécuter en temps polynomial en n.

On définit une **probabilité négligeable** de succès comme une probabilité asymptotiquement plus petite que tout inverse d'un polynôme en n.

Algorithmes PPT

Un algorithme probabiliste A est un algorithme qui peut effectuer des **choix aléatoires** (lancers de pièces deux à deux indépendants) pendant son exécution.

Un algorithme **PPT** est un algorithme **probabiliste** qui s'exécute en **temps polynomial** en la taille de l'entrée.



On considèrera que les algorithmes efficaces sont ceux qui s'exécutent en temps polynomial.

Sécurité asymptotique

Un schéma est **sécurisé** si tout adversaire PPT qui réussit à casser le chiffrement le fait avec une probabilité négligeable.

Le paramètre de sécurité permet d'ajuster le niveau de sécurité (la probabilité de succès de l'adversaire décroît exponentiellement vite).

Quelques grandeurs

Exemple Imaginons un schéma qui s'exécute en $10^6 \times n^2$ cycles et qu'un adversaire casse en $10^8 \times n^4$ cycles avec probabilité au plus $2^{-n/2}$.

Exercice Calculer les temps de calcul et la probabilité de réussite dans les deux cas suivants :

- 2 GHz CPU, n=80
- 8 GHz CPU, n=120

Augmenter la valeur de n jusqu'à obtenir le niveau de sécurité concrète souhaité.

Cryptographie à clé secrète

Un schéma de chiffrement à clé secrète est défini par 3 algorithmes :

$\mathrm{Gen}(1^n) = k$ algo probabiliste de génération de clé

$\mathrm{Enc}_k(m) = c$ algo probabiliste de chiffrement

$\mathrm{Dec}_k(c) = m$ algo déterministe de déchiffrement

$$\forall k = Gen(1^n) \quad \forall m \in \mathscr{M} \quad \mathrm{Dec}_k(\mathrm{Enc}_k(m)) = m$$

$$\forall n, \forall k = Gen(1^n) \quad |k| \geqslant n$$

Expérience d'indistinguabilité

Un adversaire A reçoit 1^n et calcule deux messages m_0 et m_1 de même longueur.

L'un de ces messages est choisi uniformément et chiffré en c par une clé k=Gen(1^n).

A doit alors deviner si le message chiffré c correspond à m_0 ou à m_1.

Un schéma est **indistinguable en présence d'une oreille indiscrète** si tous les adversaires A PPT devinent la bonne réponse avec une probabilité au plus 1/2 + fonction négligeable.

Sécurité sémantique

La notion de sécurité sémantique est équivalente à l'indistinguabilité en présence d'une oreille indiscrète.

Remarque On ne demande pas au schéma de chiffrement de cacher la taille du message chiffré. Ce peut être un problématique pour certaines applications.

Chiffrement par flot

Chiffrement par flot

S’inspire du masque jetable.

Idée : remplacer la source aléatoire du masque par un **générateur de nombres pseudo-aléatoires** bien choisi. La **racine** devient la clé.

Formellement

Un algo de chiffrement par flot est décrit par deux algorithmes déterministes :

`Init(s,IV)`, avec `s` la racine et IV un vecteur d'initialisation, calcule l'état initial st_0 ;

`GetBits(`st_i`)` calcule un (ou plusieurs) bits `y` ainsi que l'état st_{i+1} mis à jour du système.

Mise en œuvre

Générer n bits :

Entrée : s, IV, n

Sortie : $y_1, \ldots, y_n$

```
st = Init(s,IV)
for i = 1 to n:
  (y_i, st) = GetBits(st)
return y_1, ..., y_n
```

Mode synchronisé

Les deux interlocuteurs mémorisent l'état du système pour ne pas réutiliser les portions du masque déjà utilisées.

```
def start(k):
  st = Init(k)
```

```
def send(m):
  for i = 1 to |m|:
    (y_i, st) = GetBits(st)
    c_i = m_i ⊕ y_i
  return c
```

```
def recv(c):
  for i = 1 to |c|:
    (y_i, st) = GetBits(st)
    m_i = c_i ⊕ y_i
  return m
```

Adapté à un échange connecté, typiquement une unique session de communication.

Mode désynchronisé

Utiliser le vecteur d'initialisation pour éviter de réutiliser deux fois une même séquence.

```
def send(s,m):
  IV = GenIV()
  st = Init(s,IV)
  for i = 1 to |m|:
    (y_i, st) = GetBits(st)
    c_i = m_i ⊕ y_i
  return IV·c
```

```
def recv(s,IV·c):
  st = Init(s,IV)
  for i = 1 to |m|:
    (y_i, st) = GetBits(st)
    m_i = c_i ⊕ y_i
  return m
```

Sans état.

Qualité d'un chiffrement par flot

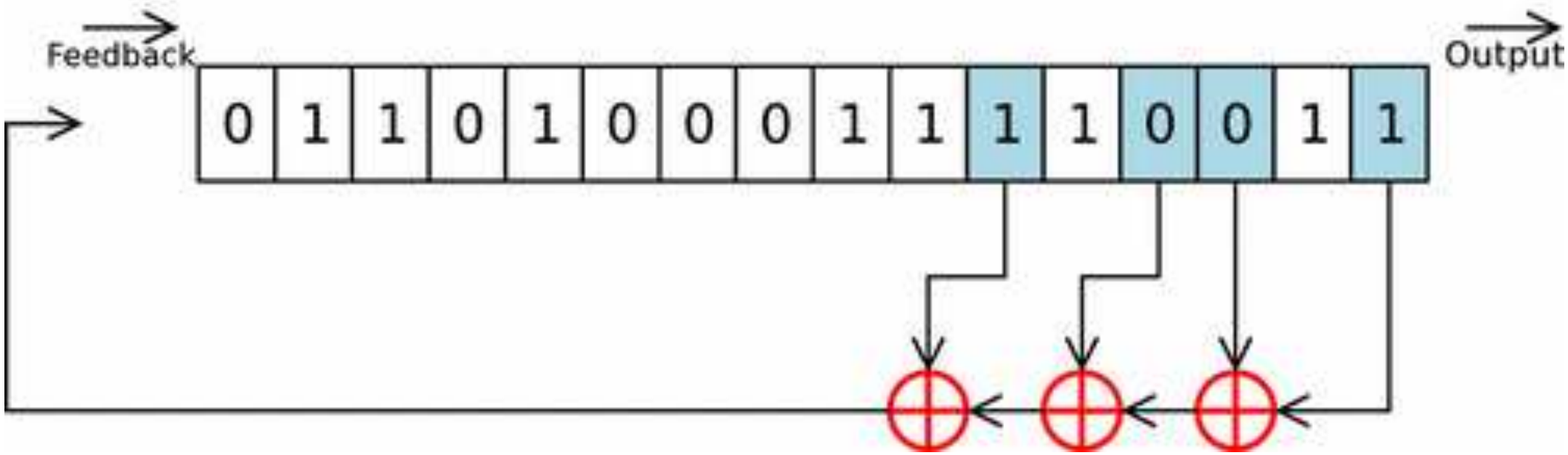
La sécurité sémantique est garantie si le générateur est pseudo-aléatoire... **de qualité cryptographique** :

- les suites de bits engendrées passent tous les tests statistiques PPT ;

- si un attaquant connait tout ou partie de la suite de bits générés par GetBits, il est difficile de retrouver la clé k utilisée (ou la paire s, IV).

LFSR : registres à décalage linéaires

Une méthode historique pour générer des nombres pseudo-aléatoires, devenue une brique de construction du chiffrement par flot.



Ajouter de la non linéarité

- en utilisant des opérateurs non linéaires ;
- en combinant plusieurs générateurs linéaires dont on contrôle les horloges ;
- *etc*

Trivium (eSTREAM 2008)

3 FSR couplés (de 93, 84, 111 bits)

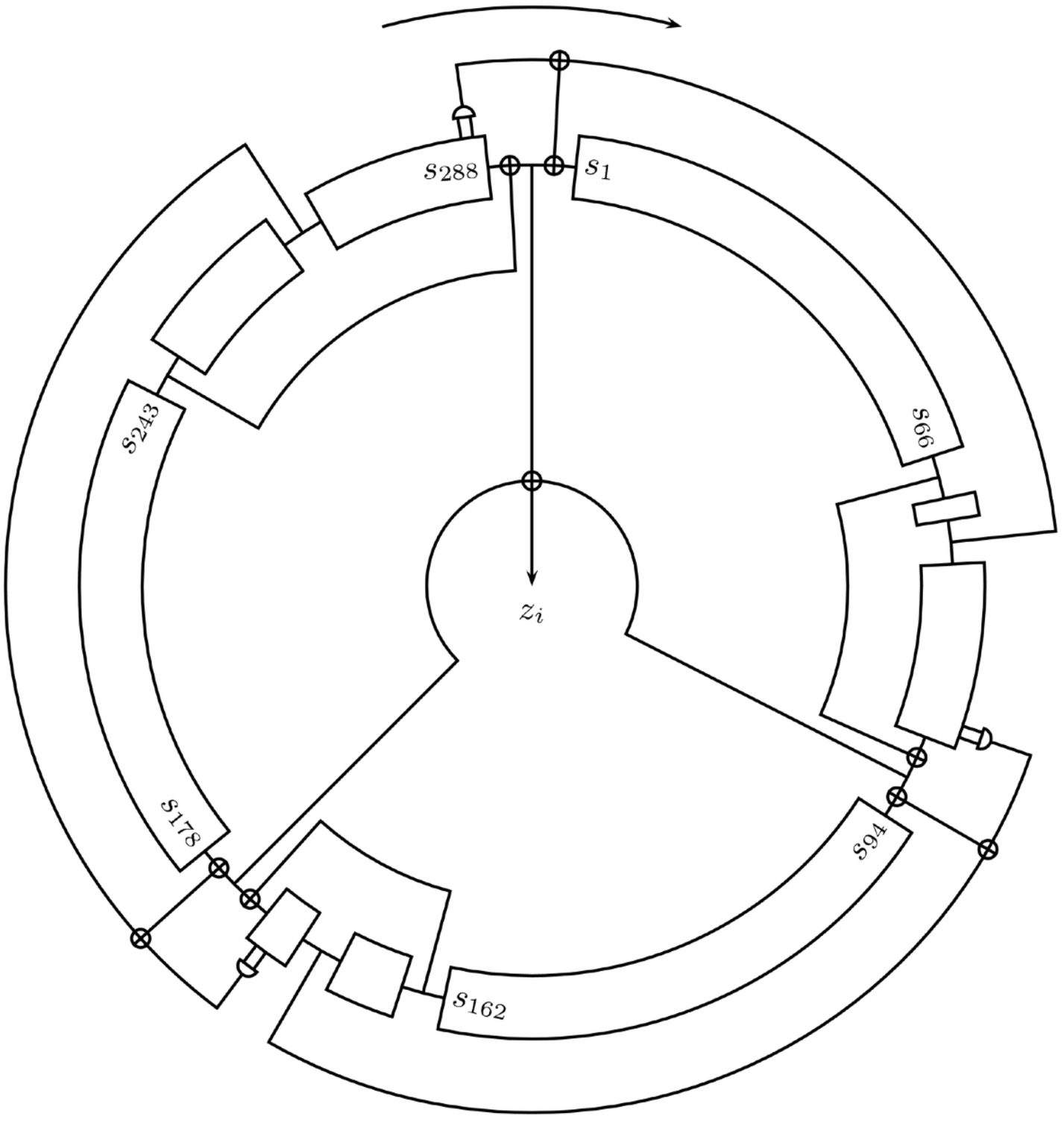
état = 288 bits

IV : 80 bits

s : 80 bits

init : remplissage

+ 4×288 itérations.



RC4

Les FSR sont compacts et rapides sous forme matérielle mais pas très efficace en logiciel.

RC4 inventé en 1987 par Ron Rivest.

A résisté de nombreuses années, attention cet algorithme **ne doit plus être utilisé**.

Lui préférer Salsa20 ou encore Chacha20.

RC4 Init

Entrée : une clé k de 16 à 256 octets
Sortie : l'état initial (S,i,j)

```
for i = 0 to 255:
  S[i] = i
n = longueur(k)
j = 0
for i = 0 to 255:
  j = (j + S[i] + k[i%n])%256
  échanger(S[i],S[j])
i = 0
j = 0
return (S,i,j)
```

RC4 GetBits

Entrée : l'état courant (S,i,j)
Sortie : un octet y et l'état mis à jour (S,i,j)

```
i = (i + 1)%256
j = (j + S[i])%256
échanger(S[i],S[j])
t = (S[i] + S[j])%256
y = S[t]
return (S,i,j), y
```

RC4 IV

L'algorithme n'a pas été conçu pour combiner un vecteur d'initialisation à la clé k.

Certains protocoles concaténent dans RC4 l'IV en tête de la clef.

Cette modification rend RC4 vulnérable à des reconstructions de la clé k à partir d'un grand nombre d'échanges désynchronisés.

Chiffrement par blocs

Principes de Shannon

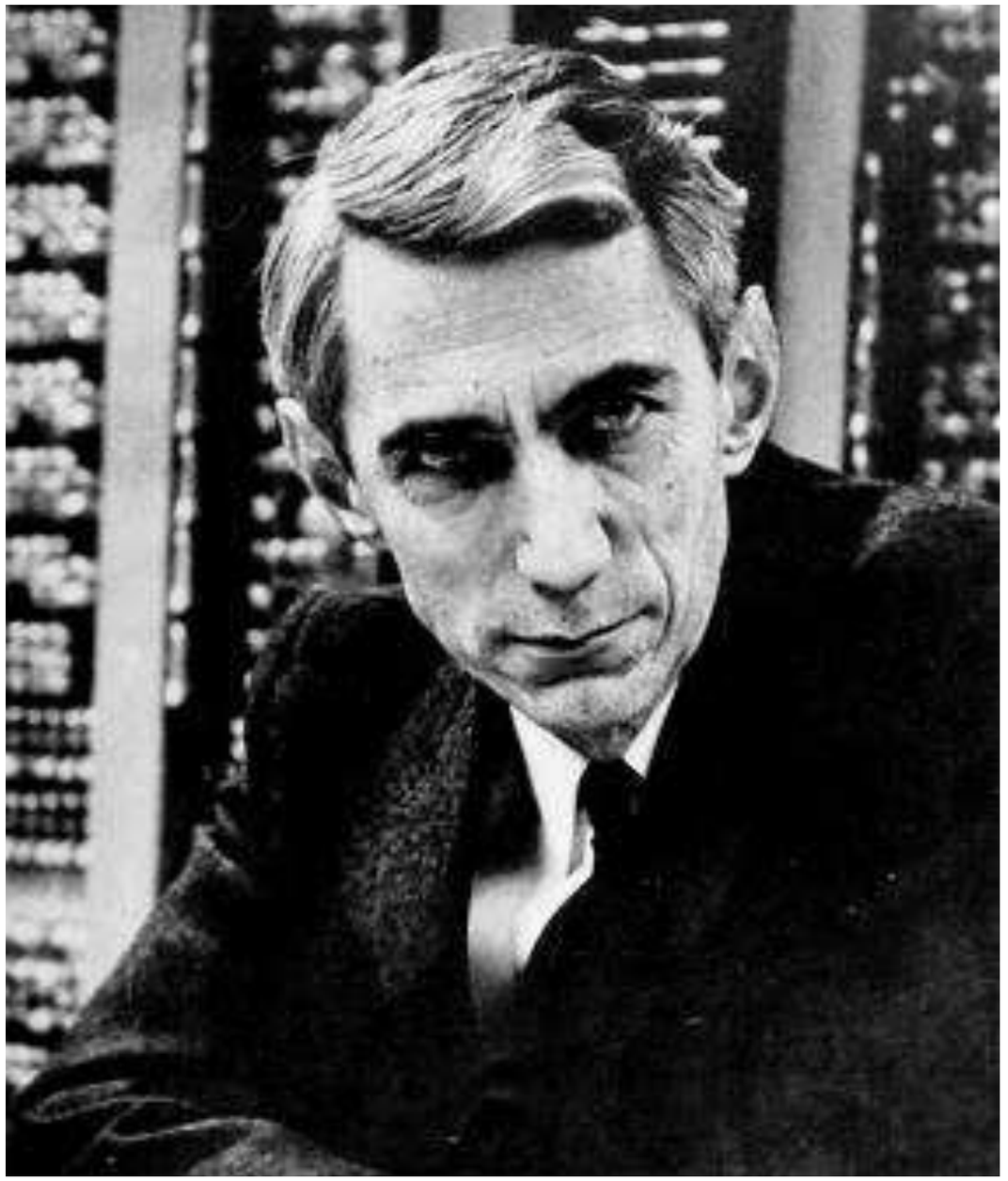
Claude Shannon (1916-2001)

Communication Theory of Secrecy Systems, Bell System Technical Journal, vol. 28(4), page 656–715, 1949.

Diffusion : mélanger l'information du message en clair dans le message chiffré.

Confusion : utiliser la clef pour camoufler le message en clair.

Effet d'avalanche : modifier un bit en entrée peut modifier tous les bits de la sortie.



Permutations pseudo-aléatoires

Plutôt que de chiffrer des messages de taille arbitraire, on se concentre sur des blocs de taille fixe.

Chaque clé possible doit générer une permutation proche de l'aléatoire.

Chiffrement par blocs

$K \in \{0,1\}^m$ clé

$M \in \{0,1\}^n$ message

$E_K : \{0,1\}^n \to \{0,1\}^n$

$D_K : \{0,1\}^n \to \{0,1\}^n$

$C = E_K(M)$ message chiffré

$M = D_K(C)$ message déchiffré

DES

Développé par IBM, adopté en 1977 par le NBS (National Bureau of Standards), le futur NIST.

DES n'est plus considéré comme sécurisé car ses clés de 56 bits sont trop courtes.

Au-delà de l'intérêt historique, il n'existe pas d'autre attaque connue que la force brute.

Chiffrement de Feistel

Idée

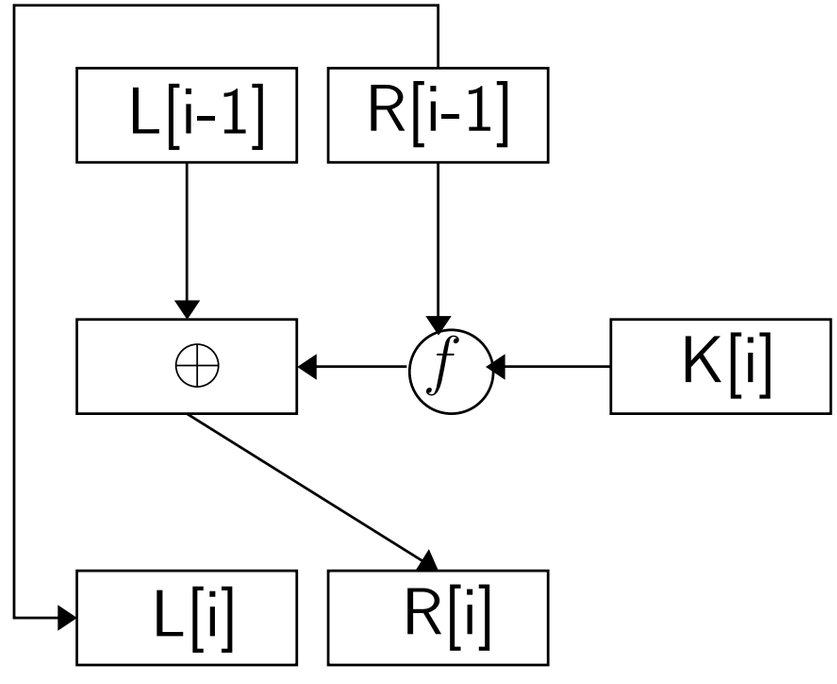
- Effectuer successivement plusieurs chiffrements simples
- La composition de chiffrements permet d'approcher une permutation quelconque
- 1973 : Feistel propose une structure générique pour les chiffrements par blocs

Chiffrement de Feistel

Input : un bloc en clair M de $2n$ bits et une clé K

M est scindé en deux messages $L[0]$ et $R[0]$ de longueurs n

L'algorithme se déroule en p étapes de mêmes structures



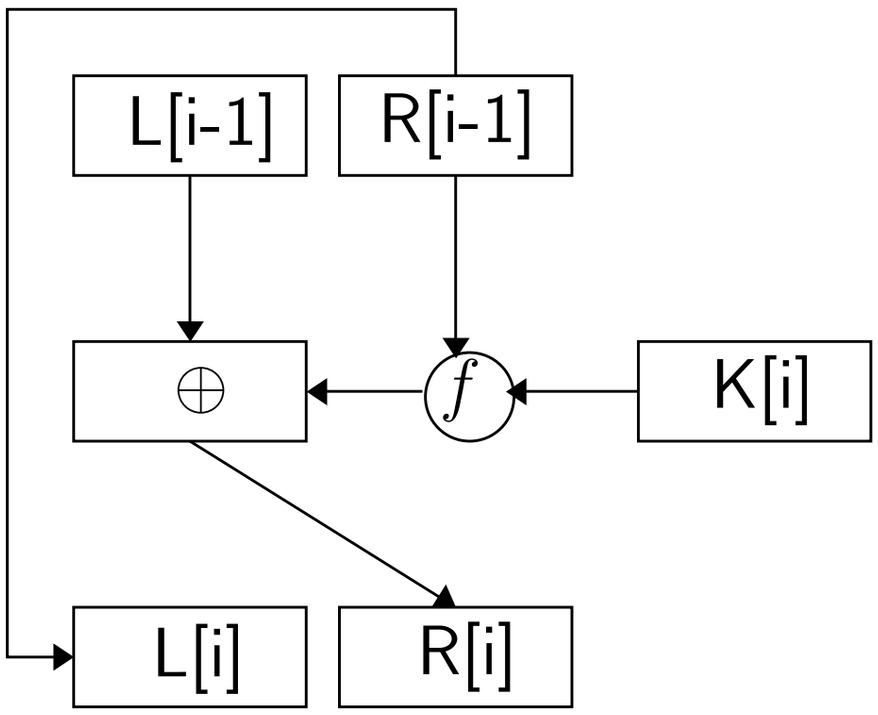
Le Chiffrement final : $L[p].R[p]$

Déchiffrement avec Feistel

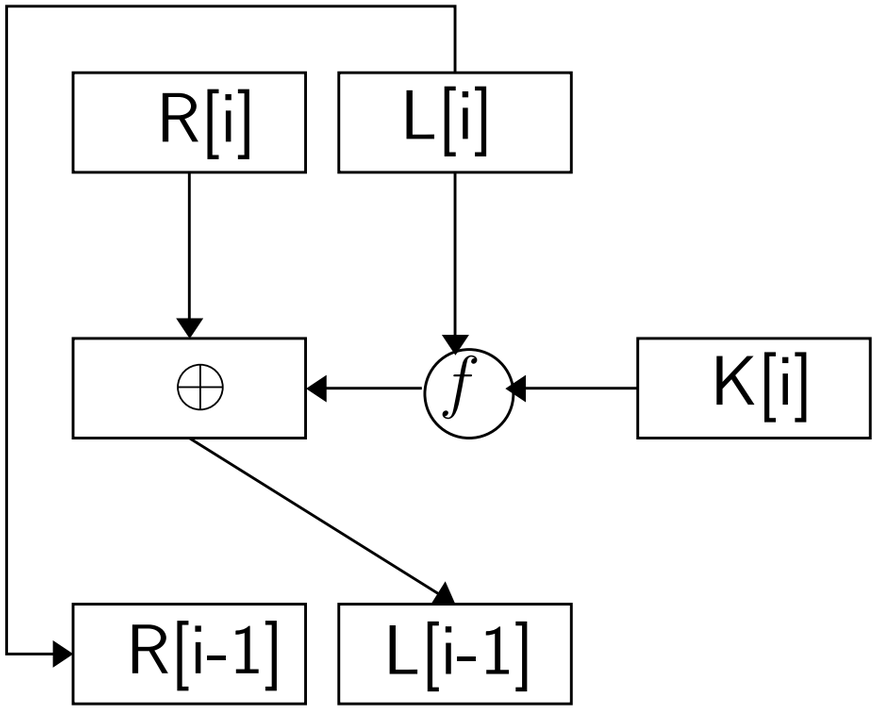
Application des clés de $K[p]$ à $K[0]$

Inversion des blocs en entrée de chaque tour

Chiffrement



Déchiffrement



DES : principe

Chiffrement par blocs de 64 bits

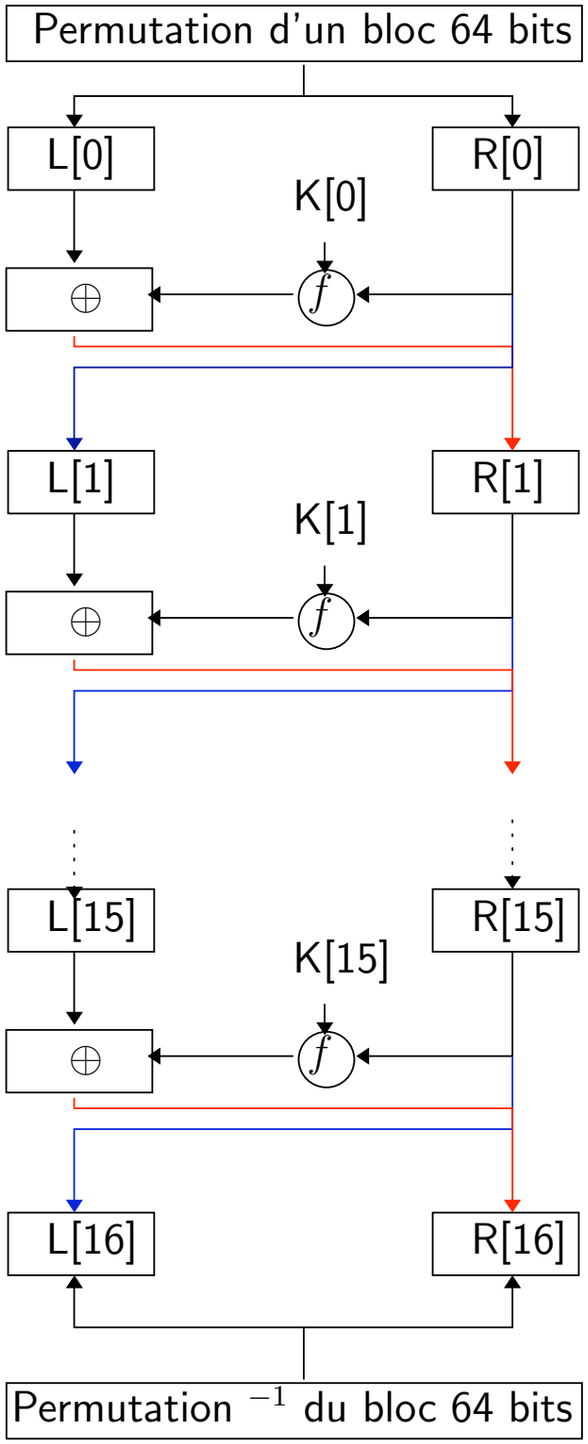
Clé de 56 bits (64 mais uniquement 56 sont utilisés)

INPUT

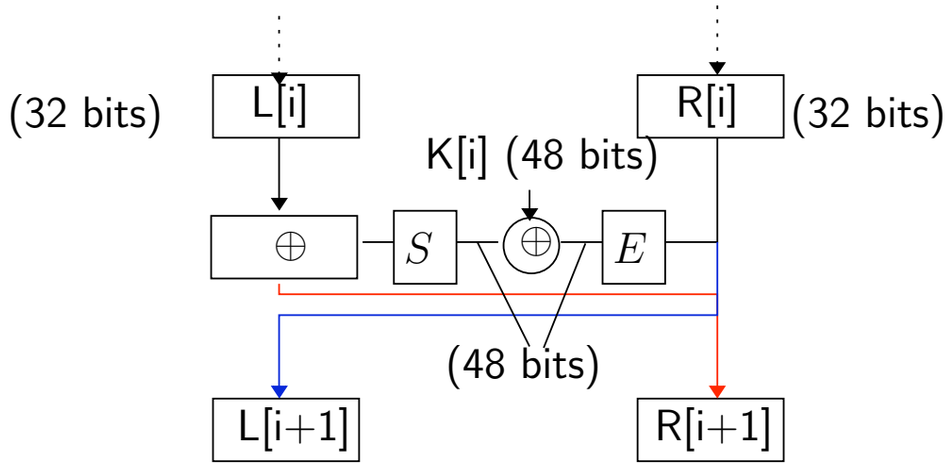
- Un bloc M de 64 bits
- Une clé K de 56 bits

DES : algorithme

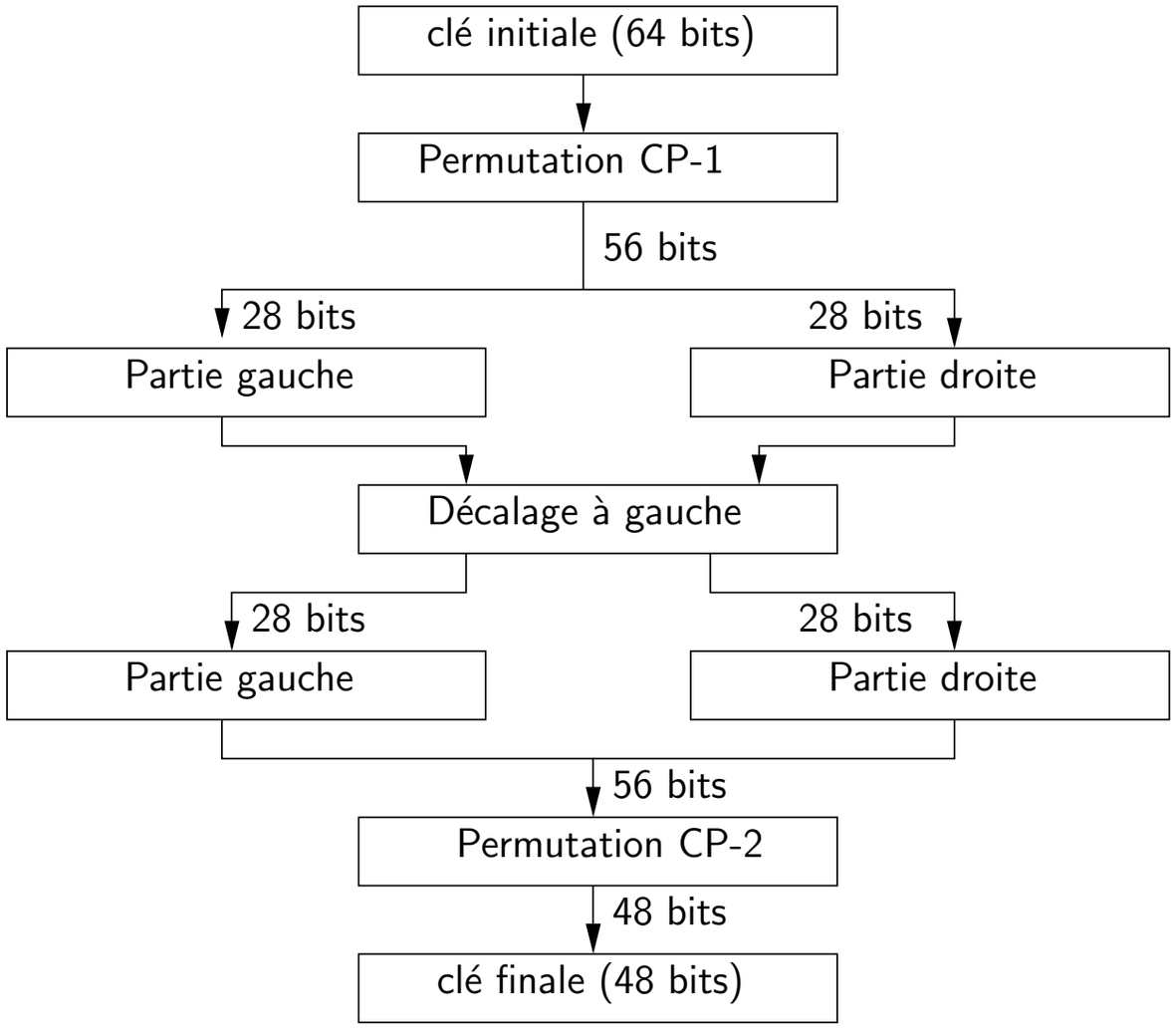
L'algorithme



Focus sur le calcul



DES : calcul des clés



CP-1

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

CP-2

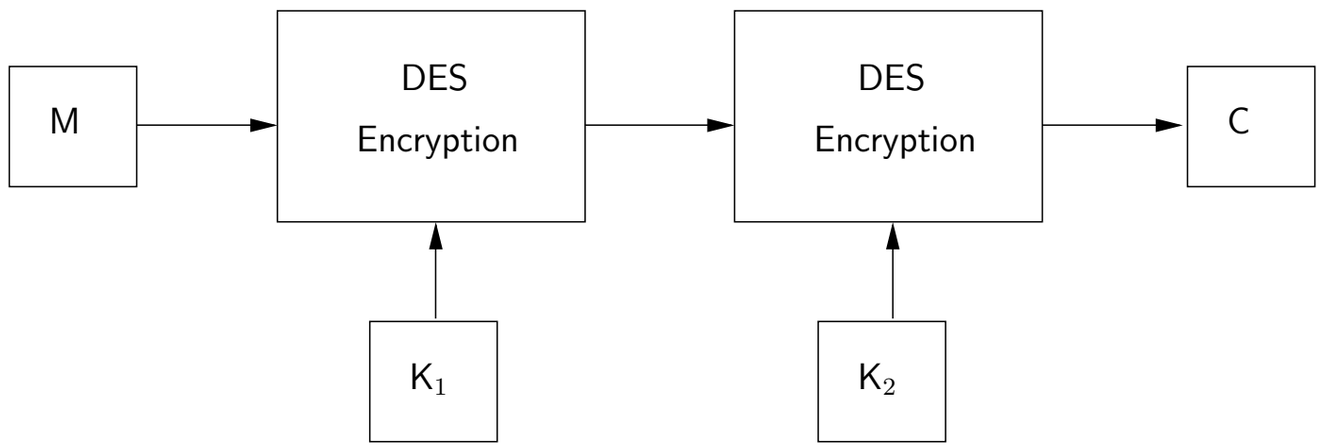
14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

DES : déchiffrement

DES est un chiffrement symétrique à clé secrète

Comme pour l'algorithme de Feistel, il faut inverser l'application des clés ainsi que les moitiés droites et gauches

DES double



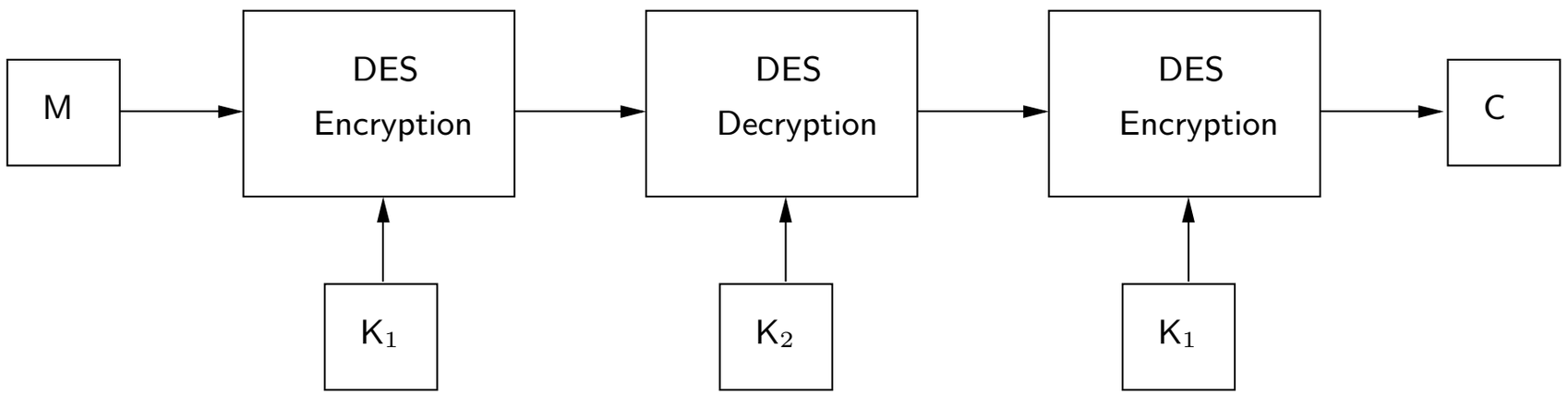
Nombre de choix possibles pour les clés : $2^{56} * 2^{56} = 2^{112}$

La sécurité du double DES $\neq$ sécurité simple DES avec une clé de 112 bits

Exercice

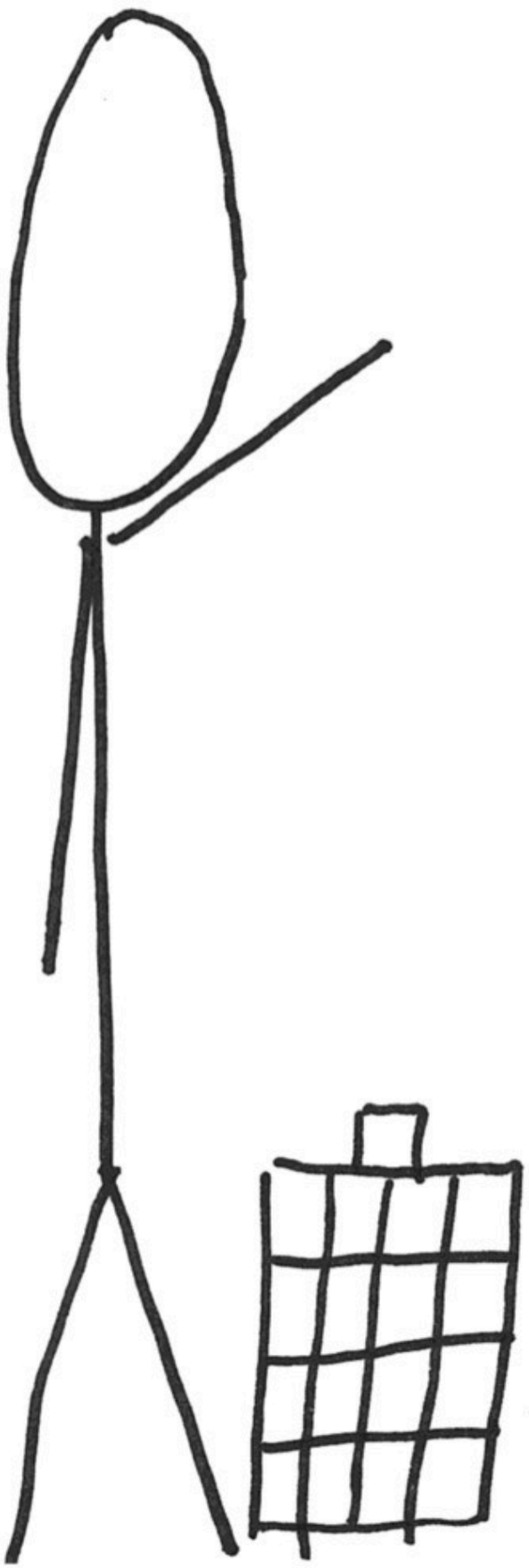
Sachant que nous avons M et C : trouvez une technique permettant de trouver K_1 et K_2 en explorant uniquement 2^{57} clés

DES triple



Intérets

- Un DES triple peut déchiffrer un message crypté en DES simple
- Puissance de calculs innaccessible actuellement



A Stick Figure Guide to the Advanced Encryption Standard (AES)