R4.B.10

Algorithmes (cryptographie)

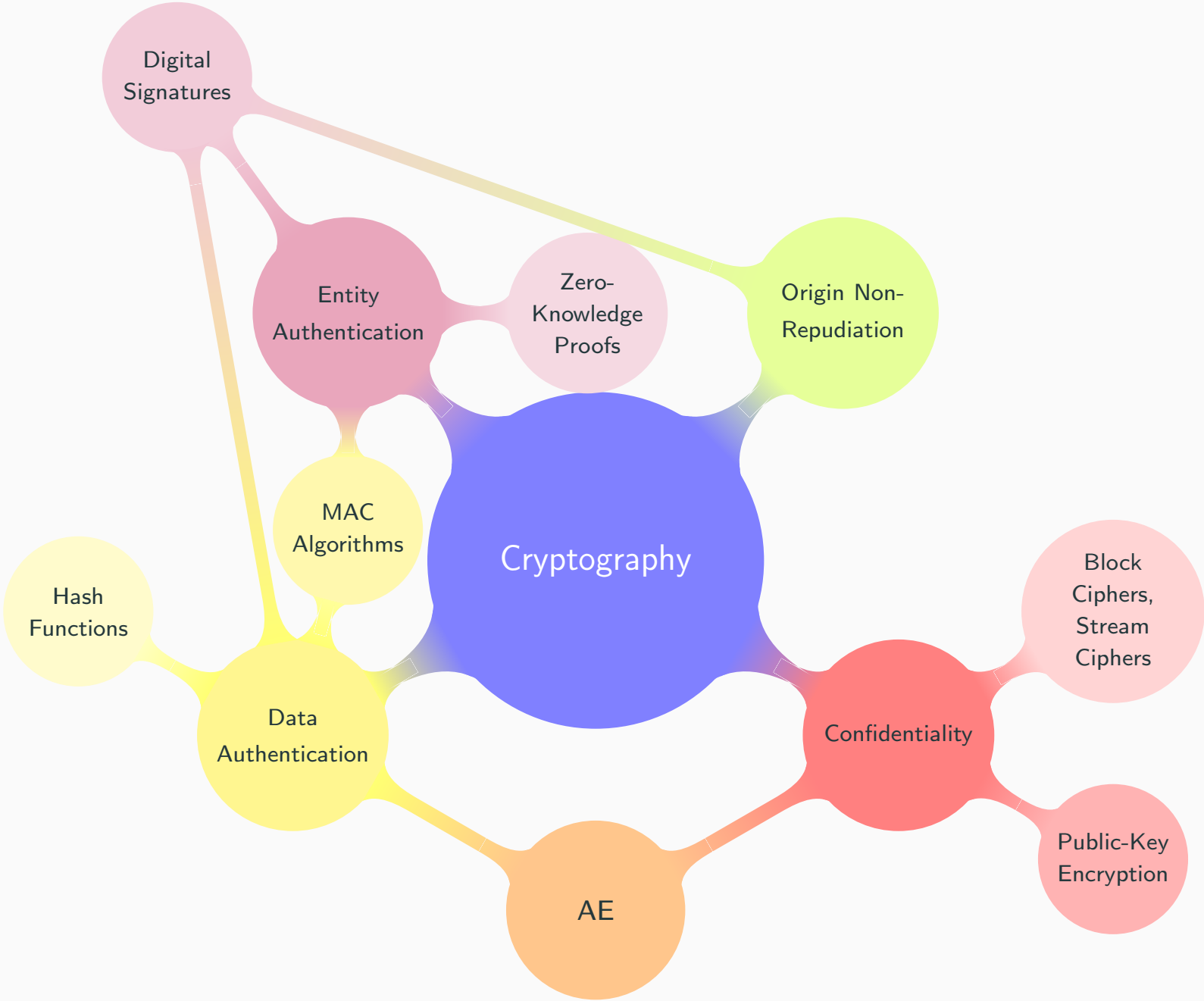
Denis Monnerat

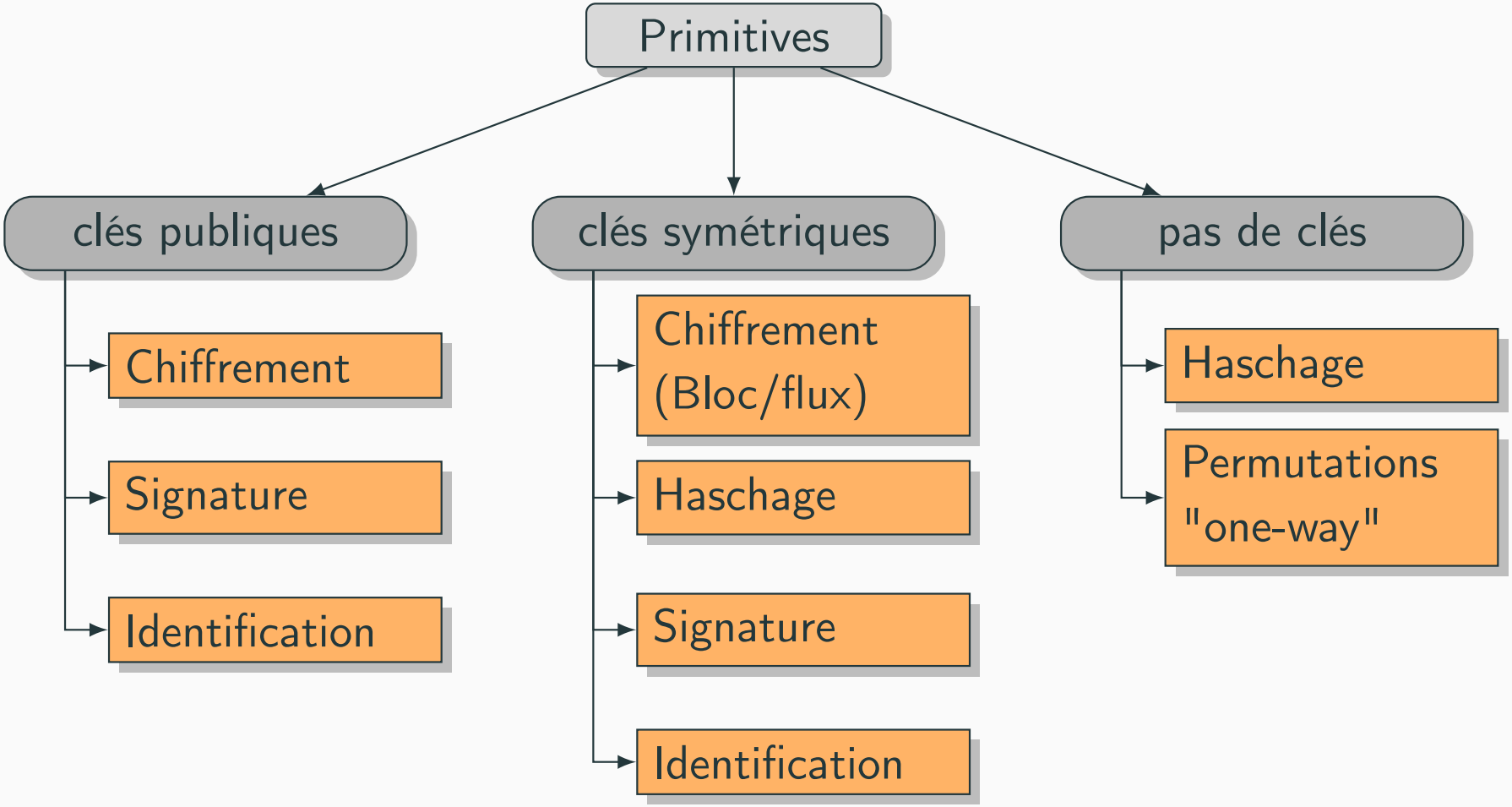
monnerat@u-pec.fr

IUT de Fontainebleau

Sommaire

Introduction

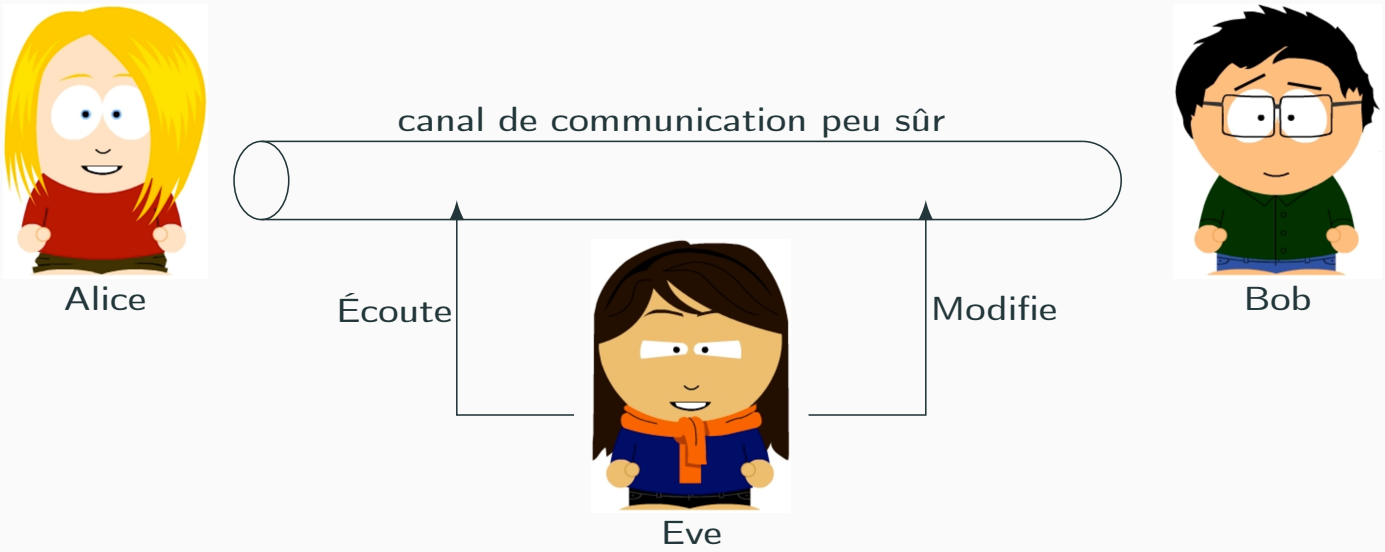

Digital Signatures
Entity Authentication
Zero-Knowledge Proofs
Origin Non-Repudiation
MAC Algorithms
Cryptography
Hash Functions
Block Ciphers, Stream Ciphers
Data Authentication
Confidentiality
AE
Public-Key Encryption


Primitives
clés publiques
clés symétriques
pas de clés
Chiffrement
Signature
Identification
Chiffrement
(Bloc/flux)
Haschage
Signature
Identification
Haschage
Permutations
"one-way"

La cryptologolie est la réunion de la cryptographie et de la cryptanalyse.

But de la cryptographie

Communiquer sur un canal peu sûr



CAIN

- Confidentialité des informations (stockées ou échangées)
- Authentification (utilisateurs, ressources)
- Intégrité
- Non répudiation (signatures)

Chiffrement/Déchiffrement

$$M \xrightarrow{E} C = E(M) \xrightarrow{D} D(C(M)) = M$$

- E fonction de chiffrement (Encryption)
- D fonction de déchiffrement (Decryption)
- E et D sont paramétrèes par des clés K_e et K_d.

Deux familles de méthodes :

1. Clés secrétes, ou symétriques $K_e = K_d$.
2. Clés publiques, ou asymétriques $K_e \neq K_d$.

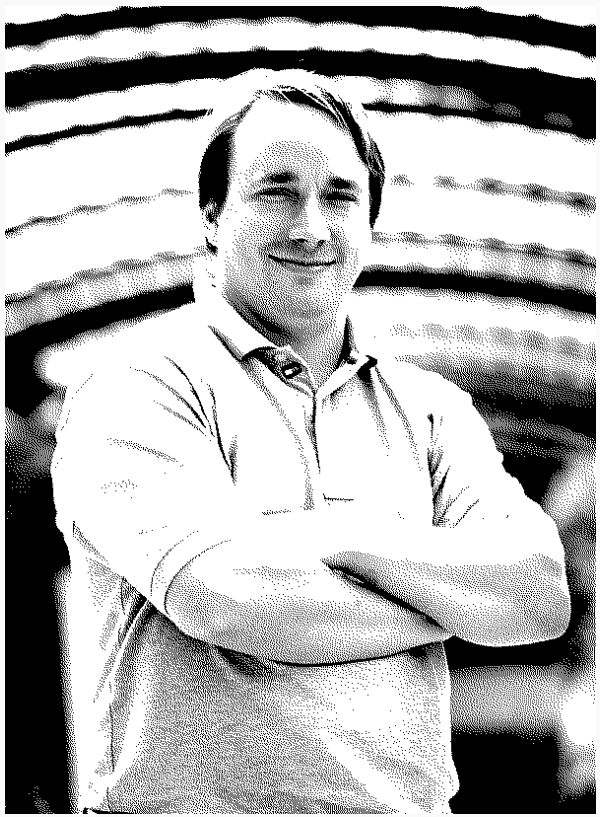
Confusion

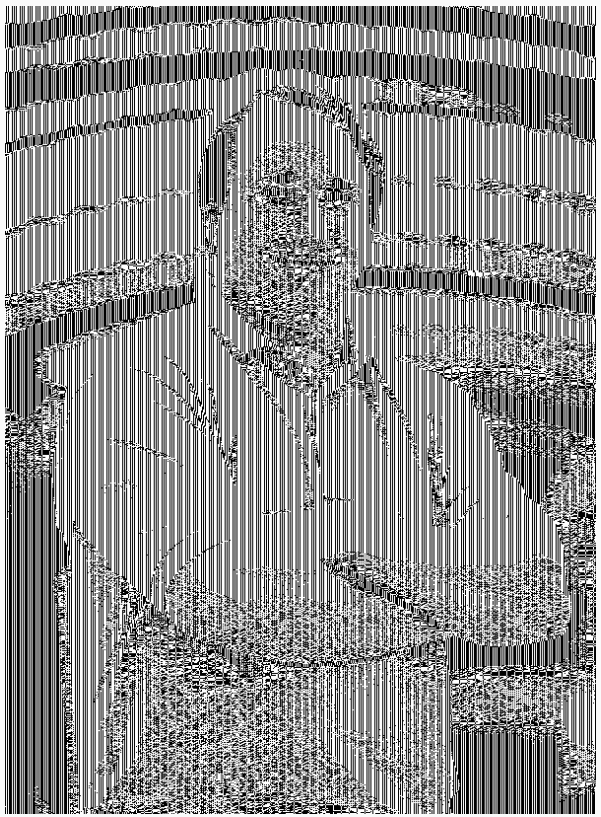
Pas de propriétés statistiques déduites du message chiffré

Diffusion

Une petite modification du message doit conduire à un message chiffré très différent.

Exemple : chaque octet o de l'image $\rightarrow o \oplus 0x8a$





Le but du cours

- Connaître les différentes définitions
- Comprendre les principes
- Utiliser des outils

Chiffrement par flux

Pas de découpage, chiffrement en continu. Quelques exemples :

- A5/1, algorithme publié en 1994, utilisé dans les téléphones mobiles de type GSM pour chiffrer la communication par radio entre le mobile et l'antenne-relais la plus proche ;
- RC4, le plus répandu, conçu en 1987 par Ronald Rivest, utilisé notamment par le protocole WEP du Wi-Fi ;
- Py, un algorithme récent de Eli Biham ;
- E0 utilisé par le protocole Bluetooth.

Souvent un chiffrement par flot se présente sous la forme d'un générateur de nombres pseudo-aléatoires avec lequel on opère un XOR entre un bit (octet) à la sortie du générateur et un bit (octet) provenant des données.
One Time Pad

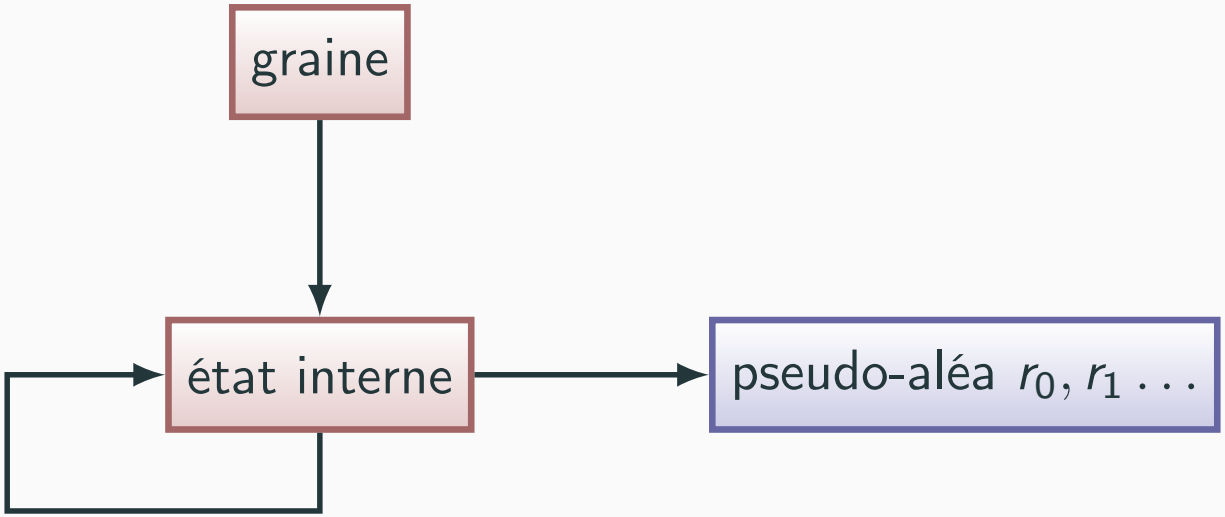
Chiffrement par flux

PRNG

Générateurs Pseudo-aléatoites

PRNG

- On part d'une graine (aléatoire).
- Cette graine alimente un état interne.
- Le générateur met à jour son état interne après chaque bit produit.



Une fois la graine spécifiée, le comportement du générateur pseudo-aléatoire est déterministe.

Usages :

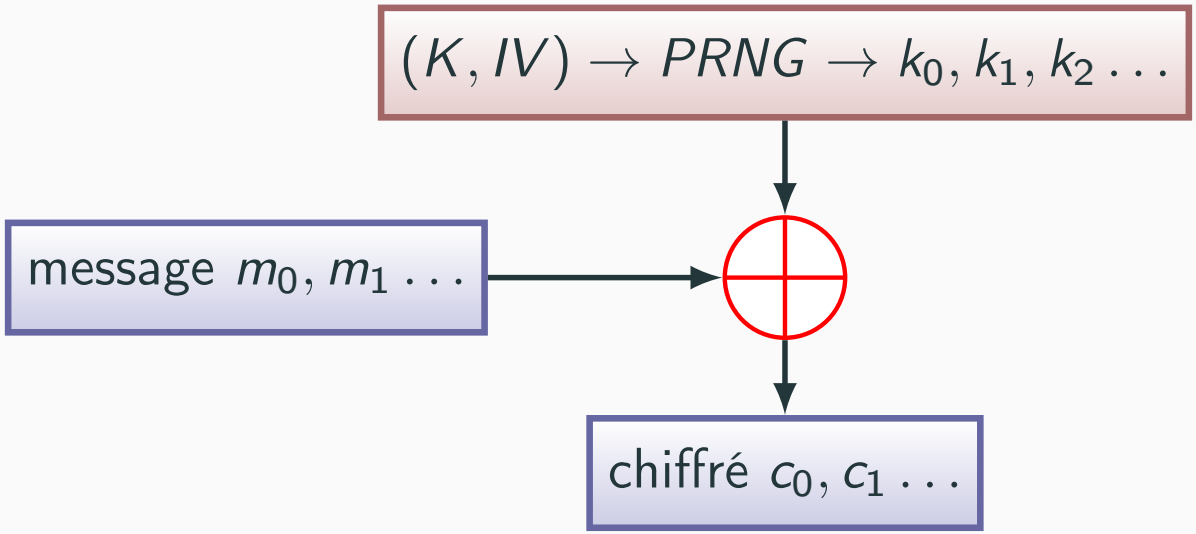
- Aléa dans les ordinateurs :
  - peu coûteux.
  - Modélisation d'événements aléatoires.
  - reproductibilité des tests.
- Crypto : utiliser une graine secrète, connue par Alice et Bob, et faire du One-Time-Pad avec.

Chiffrement par flot

⚙ difficile de générer et partager une volumineuse suite de chiffrement.

💡 utiliser un (PRNG) pour fabriquer une suite chiffrante (k_i). La graine du générateur pseudo-aléatoire sert de clé.

- Chargement de l'état interne à partir de la graine.
- Un état interne $\rightsquigarrow$ bit(s) de clé, et état interne suivant.



Un exemple de mauvais PRNG

Le `rand()` de la plupart des langages de programmation n'a pas de but cryptographique. Une version proposé par POSIX.1-2001 :

```c
static unsigned long x = 1;
/* RAND_MAX == 32767 */
int rand(void) {
  x = x * 1103515245 + 12345;
  return((unsigned)(suivant/65536) % 32768);
}
void srand(unsigned int seed) {x = seed;}
```

- Générateur linéaire congruentiel (LCG) du type

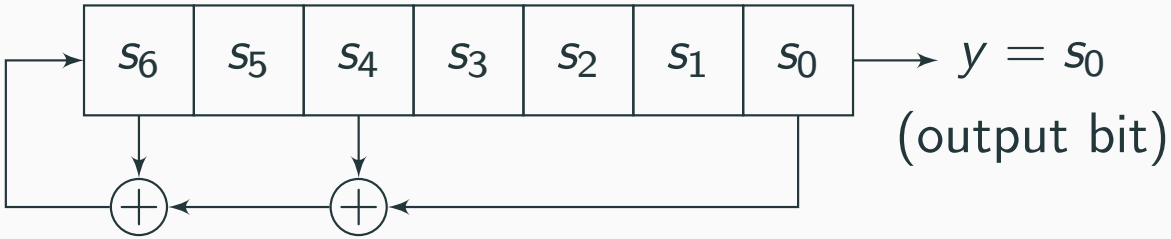
$$X_{n+1} = (aX_n + c) \mod m$$

- Facilement prédictible avec quelques valeurs successives.

Chiffrement par flux

LFSR

Linear Feedback Shift Register



$x = s_0 \oplus s_4 \oplus s_6$ (feedback bit)

L'état interne à l'étape $i+1$ dépend de l'état interne à l'étape i :

$$s_{k-1}^{(i+1)} = s_k^{(i)} \text{(shift à droite)},\ s_6^{(i+1)} = s_0^{(i)} \oplus s_4^{(i)} \oplus s_6^{(i)}$$

$\oplus$ est l'addition modulo 2, celle du corps $\mathbb{Z}/2\mathbb{Z} = \mathbb{F}_2$.

Autre expression en numérotant tous les bits :

$$\forall i \geq 0, s_{i+7} = \underbrace{s_{i+6} \oplus s_{i+4} \oplus s_i}_{\text{fonction de rétroaction}}$$

Mathématiquement

Matriciellement, en notant $S = (s_0, s_1, s_2, s_3, s_4, s_5, s_6)$, on peut écrire

$$S^{(i+1)} = S^{(i)} \times M, \text{ avec } M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

On peut interpréter la dernière colonne comme le polynôme caractéristique du LFSR (polynôme caractéristique de la matrice M)

$$P(X) = X^7 + X^6 + X^4 + 1$$

Mathématiquement

Plus généralement, un LFSR

- un entier r, taille du registre,
- un état initiale $S = (s_0, s_1, \ldots, s_{r-1}) \in \mathbb{F}_2^r$,
- une fonction linéaire f (compte la parité d'un sous-ensemble de bits),
- à chaque étape, on calcule
  $s_r = f(s_0, s_1, \ldots, s_{r-1}) = c_1 s_{r-1} + c_2 s_{r-2} + \ldots + c_r s_0$,
- à chaque top d'horloge, on décale : s_r rentre, s_0 sort,
- on obtient un nouvel état.

Exemples

$r = 4, \quad f(s_0, s_1, s_2, s_3) = s_0 \oplus s_3$

0	0	0	1
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
1	0	1	1
0	1	0	1
1	0	1	0
1	1	0	1
0	1	1	0
0	0	1	1
1	0	0	1
0	1	0	0
0	0	1	0
0	0	0	1

$r = 4, \quad f(s_0, s_1, s_2, s_3) = s_0 \oplus s_2$

0	0	0	1
1	0	0	0
0	1	0	0
1	0	1	0
0	0	1	0
0	0	0	1

L'ensemble des états générés par le registre est forcément périodique, de période $\leq 2^r - 1$.

Il faut une bonne propriété mathématique sur f pour qu'elle soit maximale ($= 2^r - 1$)

(le polynôme

$$P(X) = c_r + c_{r-1}X + c_{r-2}X^2 + \cdots + c_1X^{r-1} + X^r \in \mathbb{F}_2[X]$$

doit être primitif)

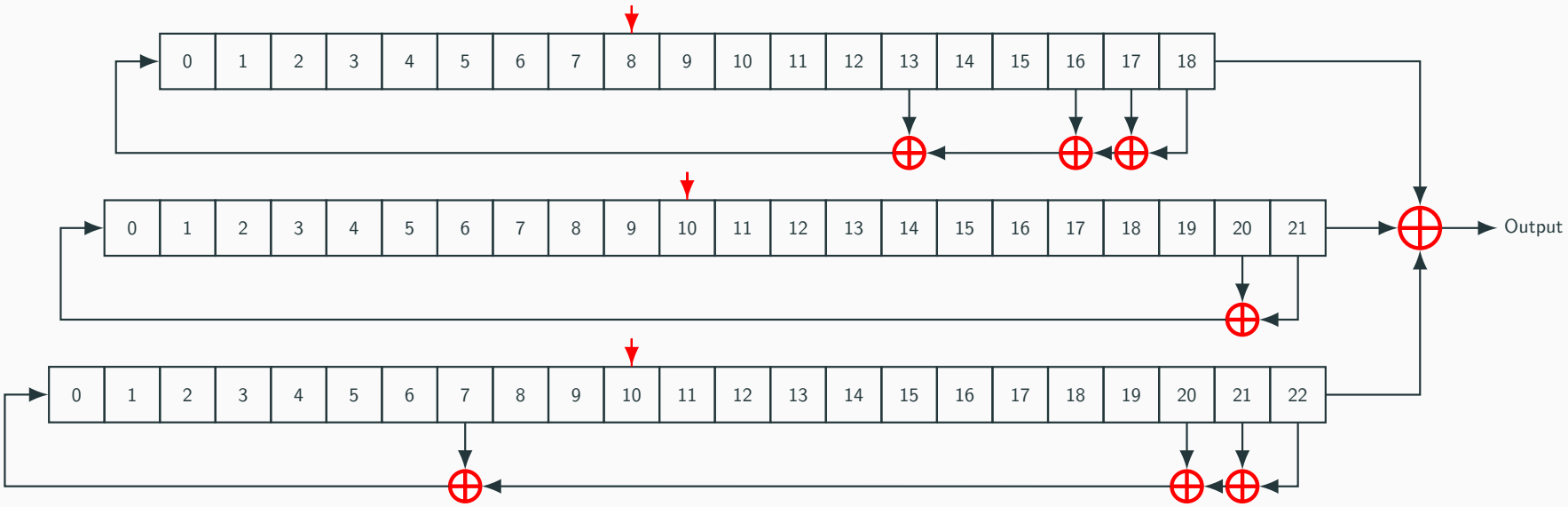
- Les LFSR ont de bonnes propriétés statistiques.
- Implantation logicielle/matérielle facile et peu onéreuse.

Malheureusement très vulnérables.

💡 Filtrer le registre avec une fonction booléenne le moins linéaire possible.

A5/1

Trois LFSR de Longueurs 19, 22 et 23, nombres premiers entre eux. Sa période est égale aux produits des périodes ($2^{19+22+23} = 2^{64}$).



Chaque registre a un cloking bit. C'est la valeur majoritaire de ces trois bits qui décident quels registres (au moins 2 donc) avancent.

Utilisation

Une conversation GSM s'effectue par division en blocs temporels, chacun dure 4,6 millisecondes et contient 2x114 bits pour les deux canaux de communication (full-duplex).

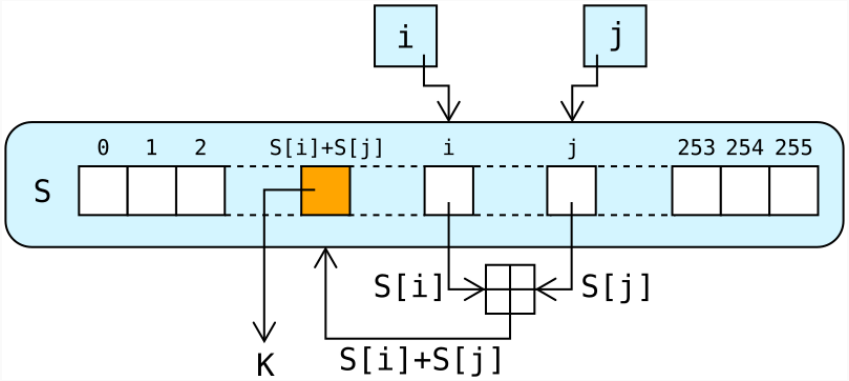
Une clé de session K est utilisée pour la communication. Pour chaque bloc, K est mélangée à un compteur de blocs et le résultat est utilisé comme état initial d'un générateur de nombres pseudo-aléatoires qui produit 228 bits. Ceux-ci servent au chiffrement après un XOR avec les données des deux canaux.

Initialisation

Tous les registres sont mis à 0.

1. On effectue 64 cycles pendant lesquels la clé de 64 bits est introduite dans le système :
   - avec le bit K_i de la clé, on effectue un XOR avec le bit de poids faible de chaque registre
   - on décale tous les registres d'un cran
2. On effectue 22 cycles pendant lesquels le numéro de trame GSM (22 bits, vecteur d'initilisation) est introduit dans le système :
   - avec le bit C_i du compteur, on effectue un XOR avec le bit de poids faible de chaque registre
   - on décale tous les registres d'un cran
3. On exécute 100 cycles supplémentaires, avec le clock majoritaire.

RC4



État interne

- un tableau S de 256 octets = permutation de $\{0, 1, \ldots, 255\}$
- 2 indices i et j dans le tableau.

La clé sert à initialiser la permutation. À chaque tour, la fonction de transition est

- $i \leftarrow (i + 1) \mod 256$
- $j \leftarrow (j + S[i]) \mod 256$
- $S[i] \leftrightarrow S[j]$

L'octet de chiffrement K est $S[(S[i] + S[j]) \mod 256]$.

Chiffrement symétrique par blocs

Chiffrement symétrique par blocs

Padding

Padding. Pourquoi ?

Problème

La taille du fichier à chiffer n'est pas toujours un multiple de la taille des blocs de l'algorithme de chiffrement

Idée : le bourrage (padding), ie on complète le fichier pour que sa taille soit un multiple exacte de la taille des blocs de l'algorithme.

Mais il faut que le déchiffrement permette sans ambiguïté de retrouver le fichier initiale non bourré.

Zero-padding

On bourre avec des octets nuls.

Exemple (bloc de 8 octets)

```
... | DD DD DD DD DD DD DD DD | DD DD DD DD 00 00 00 00 |
```

Problème : Que faire si le fichier contient des octets nuls ?

bit-padding

On rajoute dans le dernier bloc, à la fin, un bit à 1, et on compléte avec une suite (vide éventuellement) de 0.

Exemple :

```
... | 101110011101010000010011 100000000 |
```

Remarque : Si la taille du fichier est un multiple entier de la taille du bloc, on rajoutera un bloc entier

```
... | 10000000000000000000000000000000 |
```

Byte-padding

On bourre le dernier bloc, et le dernier octet vaut le nombre d'octets ajoutés. On peut bourrer avec

- des octets nuls. (ANSI X9.23 Padding)
- des octets aléatoires. (ISO 10126-2 Padding)
- le nombre d'octets ajoutés (PKCS#5 and PKCS#7).

```
... | DD DD DD DD DD DD DD DD | DD DD DD DD 00 00 00 04 |
... | DD DD DD DD DD DD DD DD | DD DD DD DD 81 A6 23 04 |
... | DD DD DD DD DD DD DD DD | DD DD DD DD 04 04 04 04 |
```

Remarque : si la taille du fichier est un multiple du bloc, on rajoute un bloc entier.

Fonctionne pour des blocs ≤ 256 octets

ISO 7816-4 Padding

Le premier octet du padding est `0x80` suivi d'octets nuls. Ce padding s'adapte facilement à n'importe quelle taille de message.

```
0x11 0x36 0x67 0x38 0xBC 0x03 0x21 0xEF | 0x80 0x00 0x00
```

Il faut partir de la fin, consommer les `00`, jusqu'à `80`.

Chiffrement symétrique par blocs

Réseaux de Feistel

Réseau de Feistel

Il s'agit d'un type d'algorithmes à clés symétriques de chiffrement par blocs. Par exemple `X(TEA)`, `DES`, `Blowfish`.

Principe

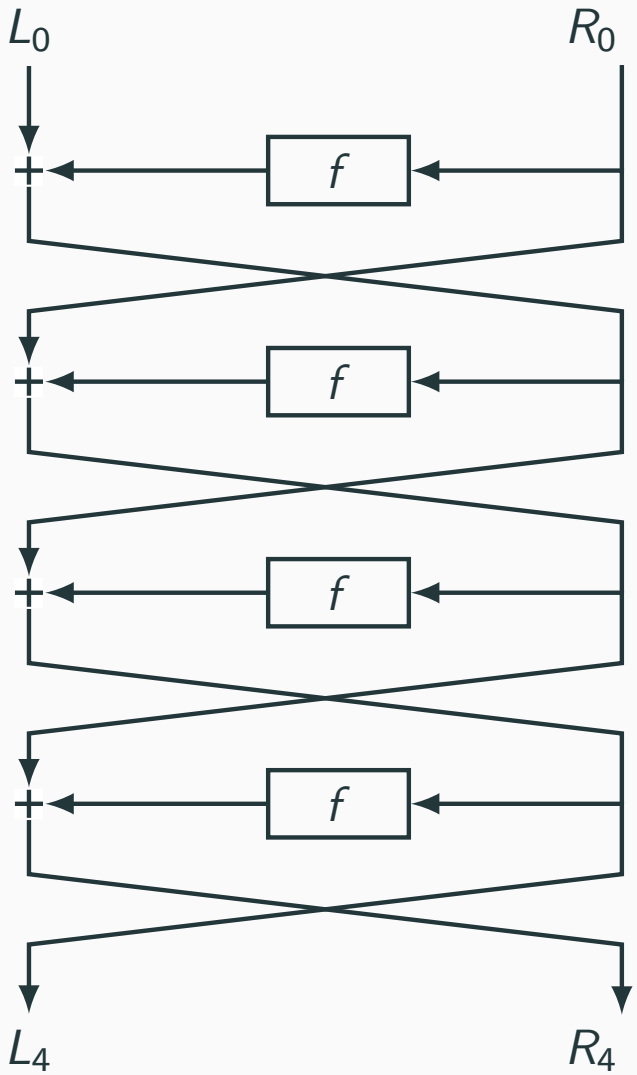
Désignons par K la clé (un mot binaire). On décompose le bloc à crypter en 2 moitiés (L_0, R_0). On lui applique une transformation de la forme

$$(L_0, R_0) \to (L_1, R_1) \text{ où } \begin{cases} L_1 &= R_0 \\ R_1 &= L_0 + f(R_0, K) \end{cases}$$

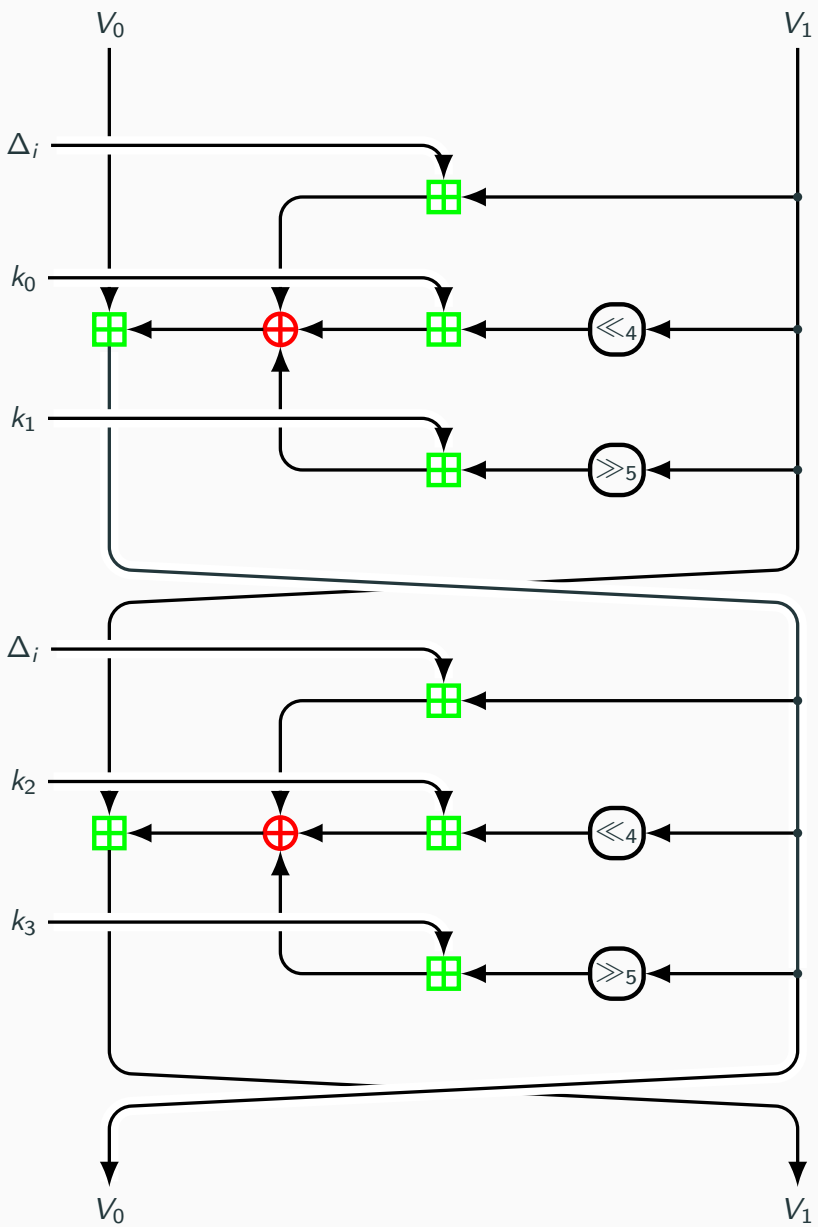
- La loi $+$ doit simplement être "réversible" (une loi de groupe). Dans la pratique, il s'agit souvent d'un xor.
- La fonction f n'a pas besoin d'être inversible pour que la transformation précédente soit réversible. Pourquoi ? Comment fait-on ?

Algorithme

Le chiffrement consiste alors à itérer la transformation (appelée round) un certain nombre de fois.



TEA



Pour TEA, la fonction de gauche (2 rounds) est itérée 32 fois.

- On découpe le bloc en deux mots de 4 octets (V_0, V_1).
- La clé est décomposée en 4 sous-clés k_0, k_1, k_2, k_3 de 4 octets.
- On utilise l'addition modulo 2^{32} $+$ et un xor $\oplus$.
- Δ_i permet d'avoir des tours non symétriques.

En C

```
void encrypt (uint32_t* v, uint32_t* k)
{
    uint32_t v0=v[0], v1=v[1], sum=0, i; /* set up */
    uint32_t delta=0x9e3779b9; /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3]; /* cache key */
    for (i=0; i < 32; i++) { /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    } /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Proposer un code pour decrypt !

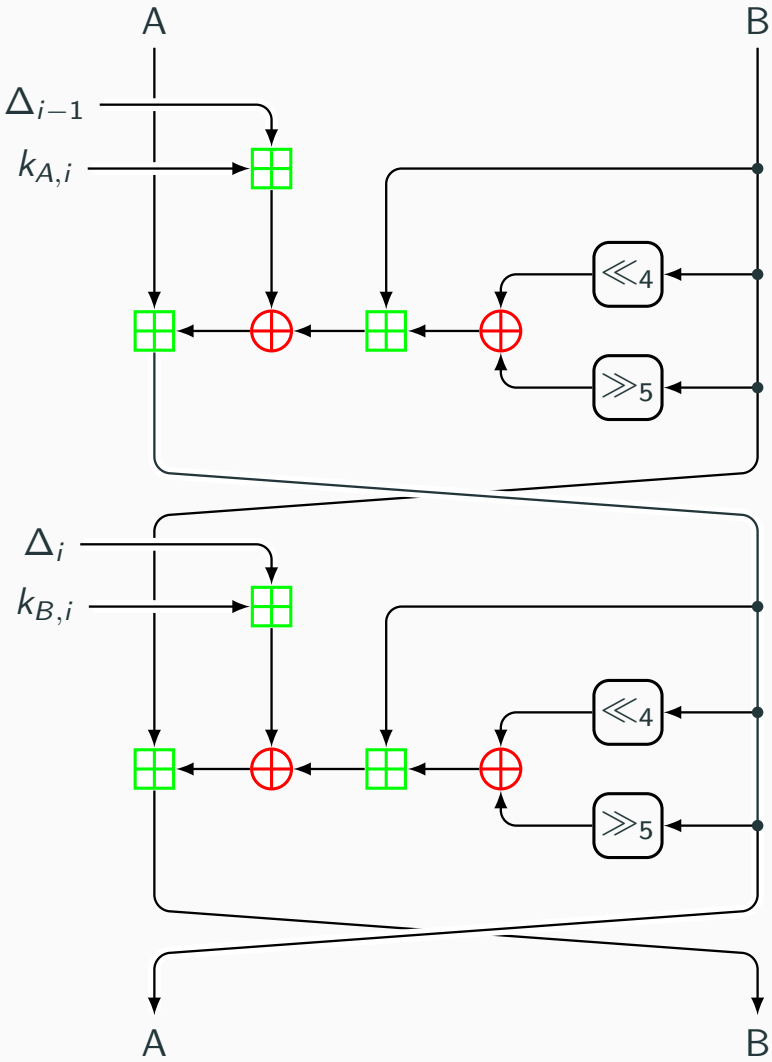
Pour être conforme à la définition d'un réseau de Feistel, on peut écrire TEA :

```
unsigned int f(uint32_t y,uint32_t k0,uint32_t k1,uint32_t delta)
{
  return ((y<< 4) + k0) ^ (y + delta) ^ ((y >> 5) + k1);
}
void encrypt (uint32_t* v, uint32_t* k)
{
  uint32_t v0=v[0], v1=v[1], sum=0, i,tmp;
  uint32_t delta=0x9e3779b9;
  uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
  for (i=0; i < 64; i++) {

    if ((i%2) == 0) {
      sum += delta;
      v0 +=  f(v1,k0,k1,sum);
    } else {
      v0 +=  f(v1,k2,k3,sum);
    }
    tmp = v0;
    v0 = v1;
    v1 = tmp;
  }
  v[0]=v0; v[1]=v1;
}
```

Variante XTEA

Blocs de 64 bits, clé de 128 bits, 64 tours recommandés. Fonction de tour



La clé k de 128 bits est découpée en 4 blocs de 32 bits.
Les sous-clefs $k_{A,i}$ et $k_{B,i}$ de chaque tour sont calculées en choisissant k_0, k_1, k_2 ou k_3 comme suit :

$k_{A,i} = k_{\Delta_{i-1}\&3}$

$k_{B,i} = k_{(\Delta_i >> 11)\&3}$

```c
void encrypt(unsigned int num_rounds, uint32_t v[2], uint32_t const key[4])
{
  unsigned int i;
  uint32_t v0=v[0], v1=v[1], sum=0, delta=0x9E3779B9;
  for (i=0; i < num_rounds; i++) {
    v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    sum += delta;
    v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum>>11) & 3]);
  }
  v[0]=v0; v[1]=v1;
}
```

Proposer un code pour `decrypt` !

Chiffrement symétrique par blocs

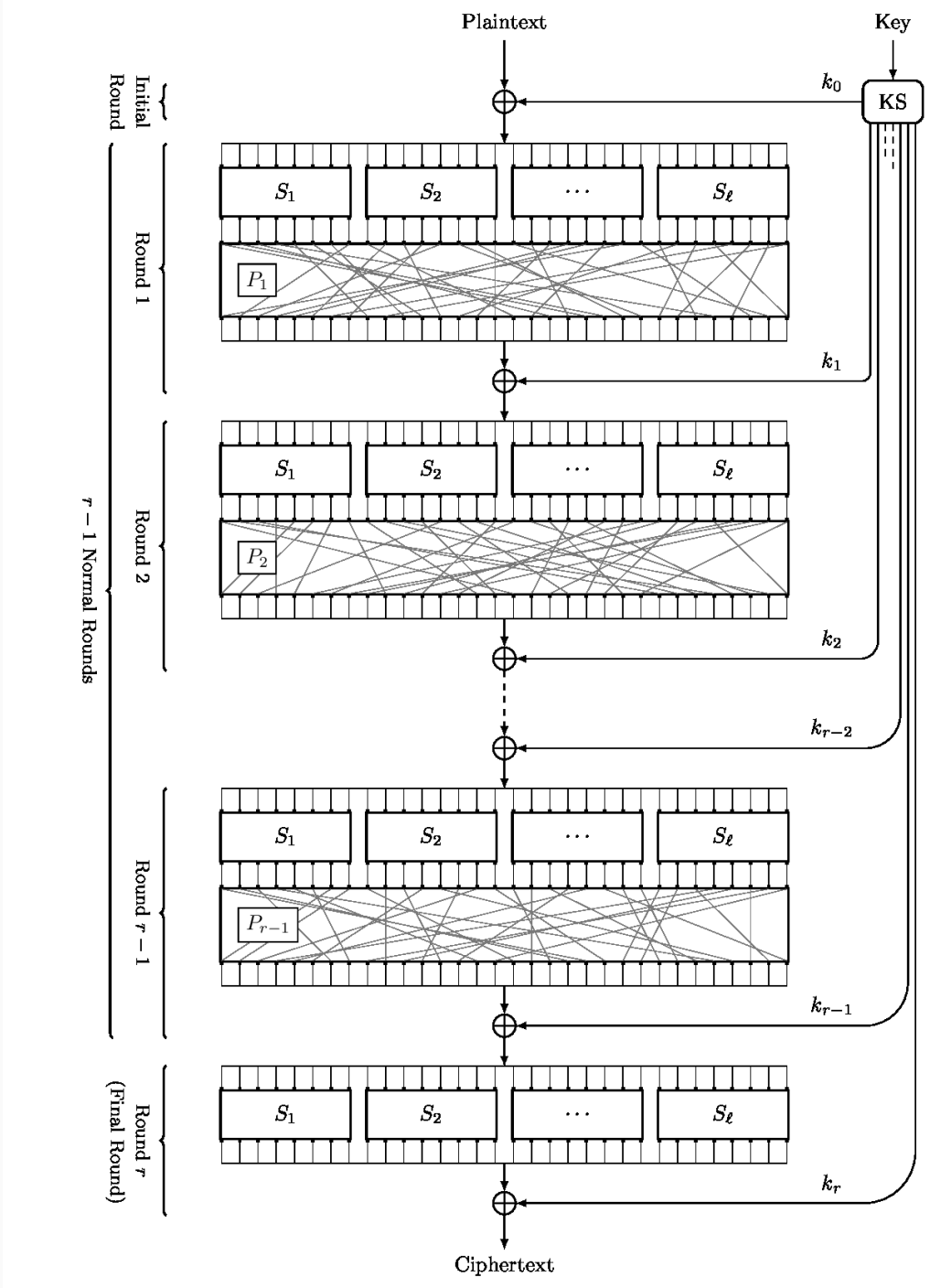
Réseaux de permutation-substitution (SPN)

Substitution-Permutation Network

Principe

Ces algorithmes utilisent en entrée une clé, et applique à chaque bloc plusieurs tours constitués de boîtes de substitutions (S-Box) et de boîtes de permutations (P-Box). Par exemple, `DES`, `AES`.

- Une S-Box substitue à un ensemble de bits un autre ensembe de bits (transormation bijective).
- Une P-Box est une permutation de bits. Elle prend la sortie d'une S-Box, permute les bits, et les transmet à une S-box au tour d'après.
- A chaque tour, on combine le bloc avec la clé (un xor souvent).


Plaintext
Key
KS
k_0
Initial Round
Round 1
S_1
S_2
$\cdots$
S_ℓ
P_1
k_1
Round 2
P_2
k_2
k_{r-2}
$r-1$ Normal Rounds
Round $r-1$
P_{r-1}
k_{r-1}
Round r (Final Round)
k_r
Ciphertext

A chaque Round :

- Un bloc de n bits en entrée, découpé en sous-blocs de l bits.
- Les substitions $S_1, \ldots, S_l$ sont des fonctions (bijectives)

$$S_i : \{0,1\}^{n/l} \to \{0,1\}^{n/l}$$

- n/l est suffisamment petit pour qu'on puisse coder une substitution dans un tableau d'entiers. Par exemple, pour $n/l = 4$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	15	2	8	7	9	14	4	11	6	3	5	12	10	0	13

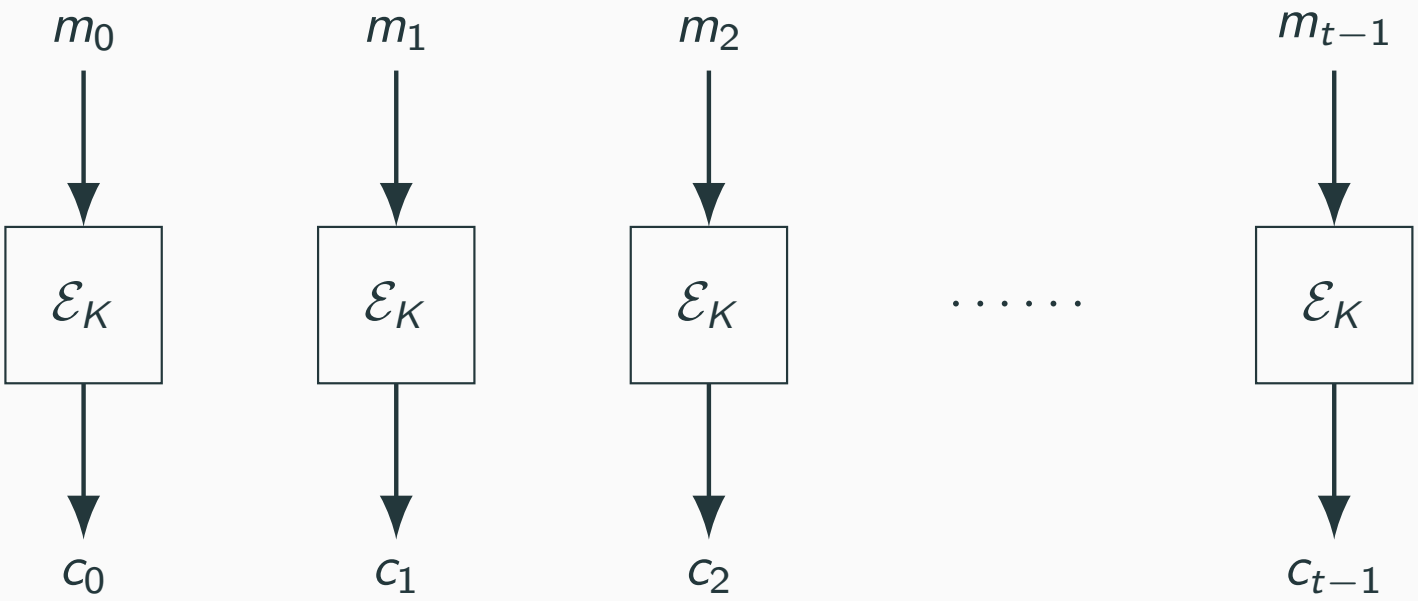
0111 (7) est substitué par 0100 (4)

Chiffrement symétrique par blocs

Modes ECB, CBC, CTR, OFB

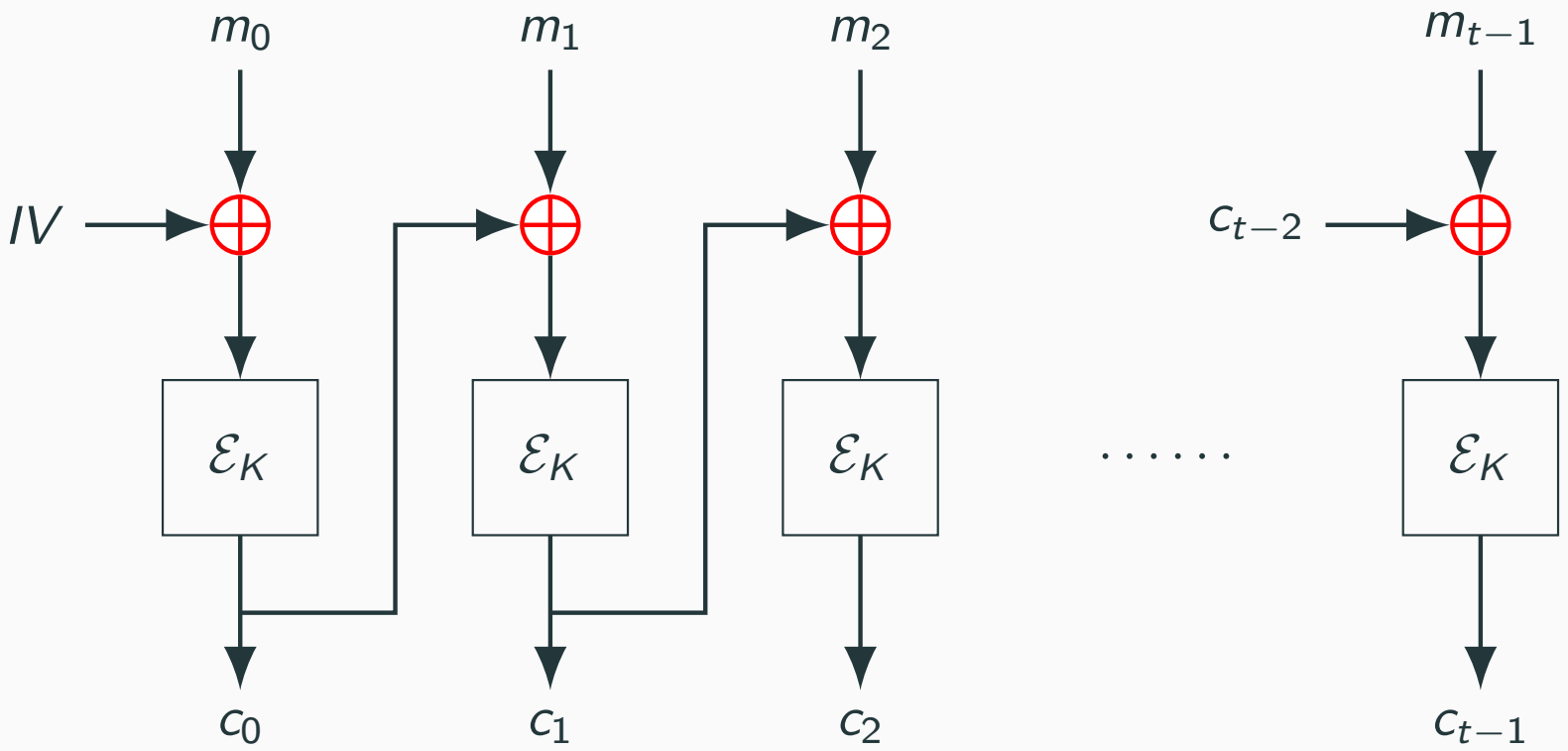
Mode ECB (Electronic Code Book)

Les blocs sont chiffrés indépendaments. 2 blocs identiques ont le même chiffré :-(

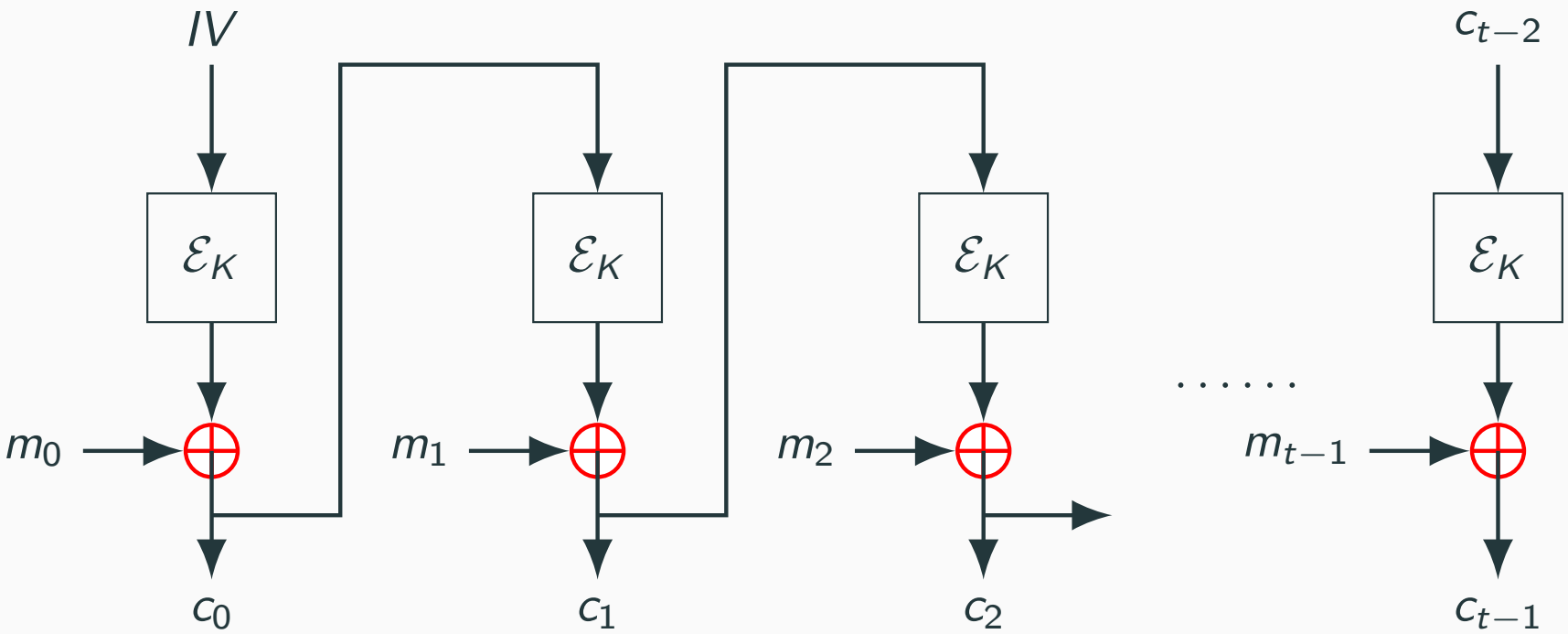


Mode CBC (Cipher Block Chaining)

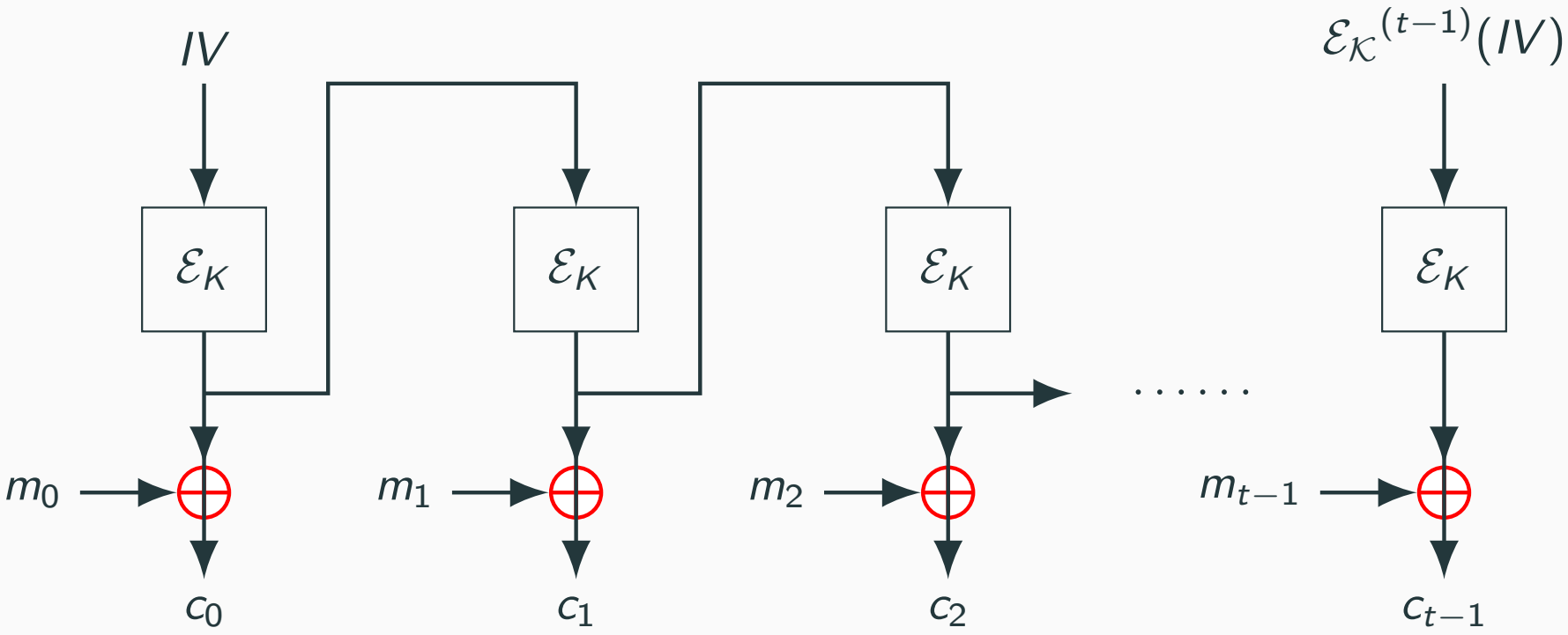
Utilisation d'un vecteur d'utilisation (IV). Non prédictible et unique. Pas besoin d'être secret (pourquoi ?)



Mode CFB (Cipher Feed Bloc)

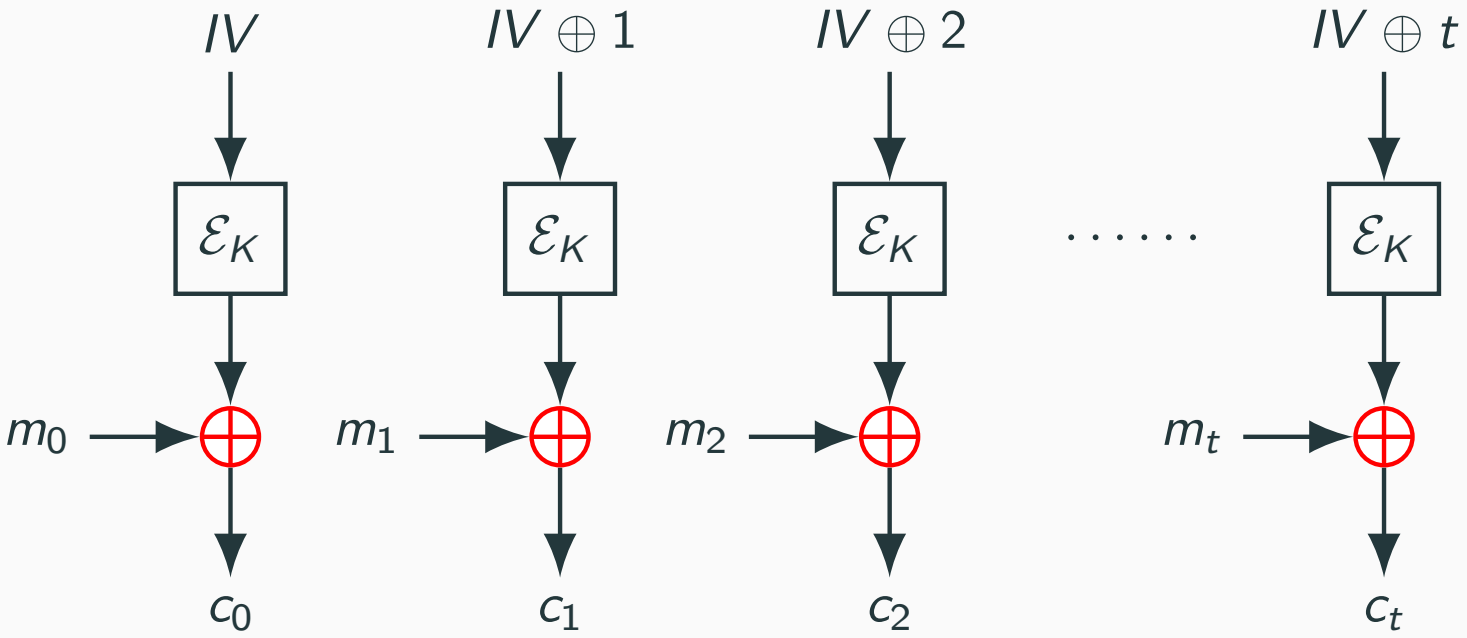


Mode OFB (Output Feedback Mode)



Mode CTR (Counter)

Permet d'utiliser un chiffrement par block en mode flux.



Chiffrement symétrique par blocs

Malléabilité

Malléabilité

Possibilité de modifier le message chiffré, sans le déchiffrer, même en mode CBC.

Un exemple : supposons que l'on chiffre avec AES-CBC le message

"Your Password is totototatatototat".

Le chiffré consiste en $IV||C_0||C_1$ calculé avec une clé K, avec $M_0 =$ "your password is" et $M_1 =$ " totototatatototat".

$$\begin{aligned} C_0 &= E_K(M_0 \oplus IV) \\ C_1 &= E_K(C_0 \oplus M_1) \end{aligned}$$

Imaginons que M_0 soit connu, on peut injecter facilement 2 blocs quelconques B_0 et B_1 en forgeant

$$\begin{aligned} X_0 &= IV \oplus M_0 \oplus B_0 \\ X_1 &= IV \oplus M_0 \oplus B_1 \end{aligned}$$

Et en envoyant la concaténation $X_0||C_0||X_1||C_0||C_1$, calculez le message déchiffré ? Pourquoi est-ce embêtant ?

MAC

Message Authentification Code.

- Intégrité des données
- Différent de la confidentialité.

Mac est calculé comme le dernier bloc d'un chiffrement CBC.

- Chiffrement CBC du message, on ne garde que le dernier chiffré $\equiv MAC$.
- Somme de contrôle cryptographique.

- Calcul (Message de N blocks : $M_0, M_1, \ldots, M_{N-1}$) :

$$\begin{aligned} C_0 &= E(IV \oplus M_0, K) \\ C_1 &= E(C_0 \oplus M_1, K) \\ C_2 &= E(C_1 \oplus M_2, K) \\ \ldots &= \ldots \\ C_{N-1} &= E(C_{N-2} \oplus M_{N-1}, K) = MAC \end{aligned}$$

- On envoie $IV, M_0, M_1, \ldots, M_{N-1}, MAC$.
- Le destinataire fait le même calcul, et vérifie le MAC reçu.
- La clef est un secret partagé.

Est-ce que ça fonctionne ?

Un exemple :

- Alice envoie à Bob 4 blocs.
- Alice calcule :

$$\begin{aligned} C_0 &= E(IV \oplus M_0, K) \\ C_1 &= E(C_0 \oplus M_1, K) \\ C_2 &= E(C_1 \oplus M_2, K) \\ C_3 &= E(C_2 \oplus M_3, K) = MAC \end{aligned}$$

- Alice envoie $IV, M_0, M_1, M_2, M_3, MAC$.
- Eve change M_1 en X.
- Bob calcule : $C_0 = E(IV \oplus M_0, K)$, $C_1 = E(C_0 \oplus X, K)$, $C_2 = E(C_1 \oplus M_2, K)$, $C_3 = E(C_2 \oplus M_3, K) = MAC \neq MAC$
- la modification se propage jusqu'à MAC. Il faudrait qu'Eve calcule le bon MAC, mais sans connaître K ...

Fonctions de hachage

Fonctions de hachage

Fonctions de Hachage

Fonction de hachage

Définition

Une fonction de hachage h de longueur n calcule à partir d'un message m une empreinte (un condensé) $h(m)$ de longueur n.

$$h : \begin{cases} \bigcup_{k \in \mathbb{N}} \{0,1\}^k & \longrightarrow & \{0,1\}^n \\ m & \longrightarrow & h(m) \end{cases}$$

- $h(m)$ est appelé empreinte, résumé, condensé, etc.
- une telle fonction h par nature ne peut pas être injective. Pourquoi ?
- une fonction de hachage n'est pas une fonction de chiffrement !

Exemple : `MD2`, `MD3`, `MD4`, `MD5`, `MD6` `SHA-0`,`SHA-1`, `SHA-2`, `SHA-3`

BUT

Intégrité des données

- Vérifier que les données transmises n'ont pas subi d'altérations.
- Produire une empreinte qui soit facilement vérifiable.

Propriétés

Une bonne fonction de hachage doit être :

- publique (pas de notion de secret),
- facile et rapide à calculer,
- à sortie de taille fixe (quelque soit l'entrée),
- bien répartie en sortie ("mélange" bien).

Propriétés

Résistance aux collisions

Il doit être difficile de trouver m_1 et m_2 tels que $h(m_1) = h(m_2)$

Résistance à la préimage

Étant donnée e, il doit être difficile de trouver m tel que $h(m) = e$

Résistance à la seconde préimage

Étant donnée x, il doit être difficile de trouver $y \neq x$ tel que $h(x) = h(y)$

Rapidité

Le calcul $h(m)$ doit être rapide.

Diffusion

Une petite modification du message m doit entraîner une grande modification de l'empreinte $h(m)$.

Exemple avec MD5

```
$echo -n "toto" |md5sum
f71dbe52628a3f83a77ab494817525c6
$echo -n "toto!" |md5sum
e8be07df544a3d4ffdfb755759309618
```

Priviliégier SHA-2 (SHA-512) ou SHA-3

```
echo -n "toto" | openssl dgst -sha512
10e06b990d44de0091a2113fd95c92fc905166a
f147aa7632639c41aa7f26b1620c47443813c60
5b924c05591c161ecc35944fc69c4433a49d10fc6b04a33611
```

Applications

- Vérification de l'intégrité des fichiers ou des messages.
- Vérification de mot de passe.
- Identification de fichiers ou de données.

Paradoxe des anniversaires

Problème

Calculer le nombre de personnes que l'on doit réunir pour avoir une chance sur deux que deux personnes de ce groupe aient leur anniversaire le même jour.

Résultat

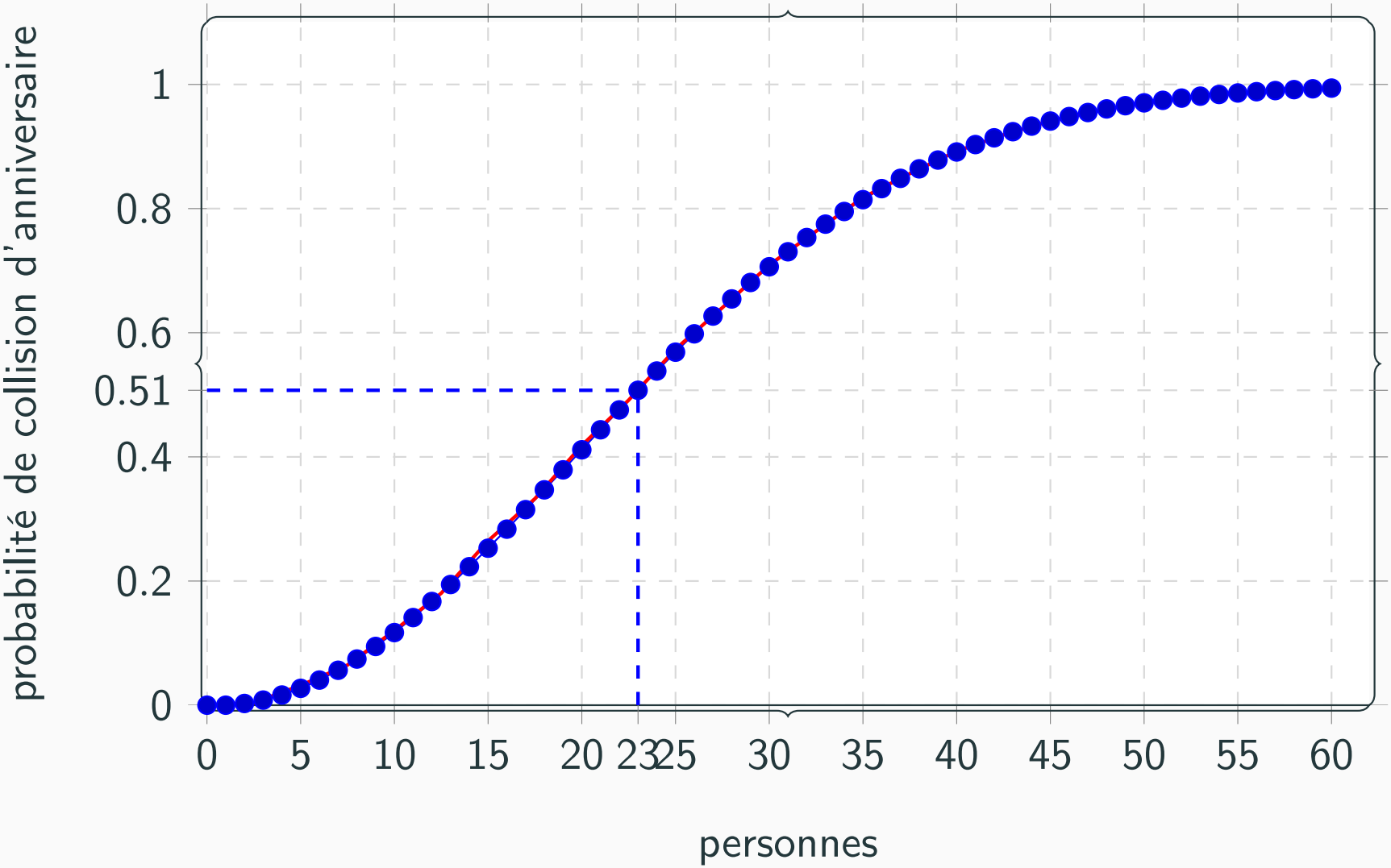
Une classe de 23 des élèves est suffisante, ce qui choque l'intuition, car il y a 365 possibilités pour les dates d'anniversaire !

- $P = P($ deux personnes nées le même jour$)$
- $\bar{P} = P($aucune de personnes néees le même jour$)$

n personnes $\Rightarrow 356^n$ possibilités. Si on veut des jours différents, nous obtenons un arragement de n parmi 356 :

$$A_{365}^n = (365-0)(365-1)\dots(365-n+1) = \frac{365!}{(365-n)!}.$$

$$\bar{P} = \frac{1}{365^n}\frac{365!}{(365-n)!} \text{ et } P = 1 - \bar{P}$$



À partir de 57 personnes, la probabilité est supérieure à 99%.

Attaque des anniversaires

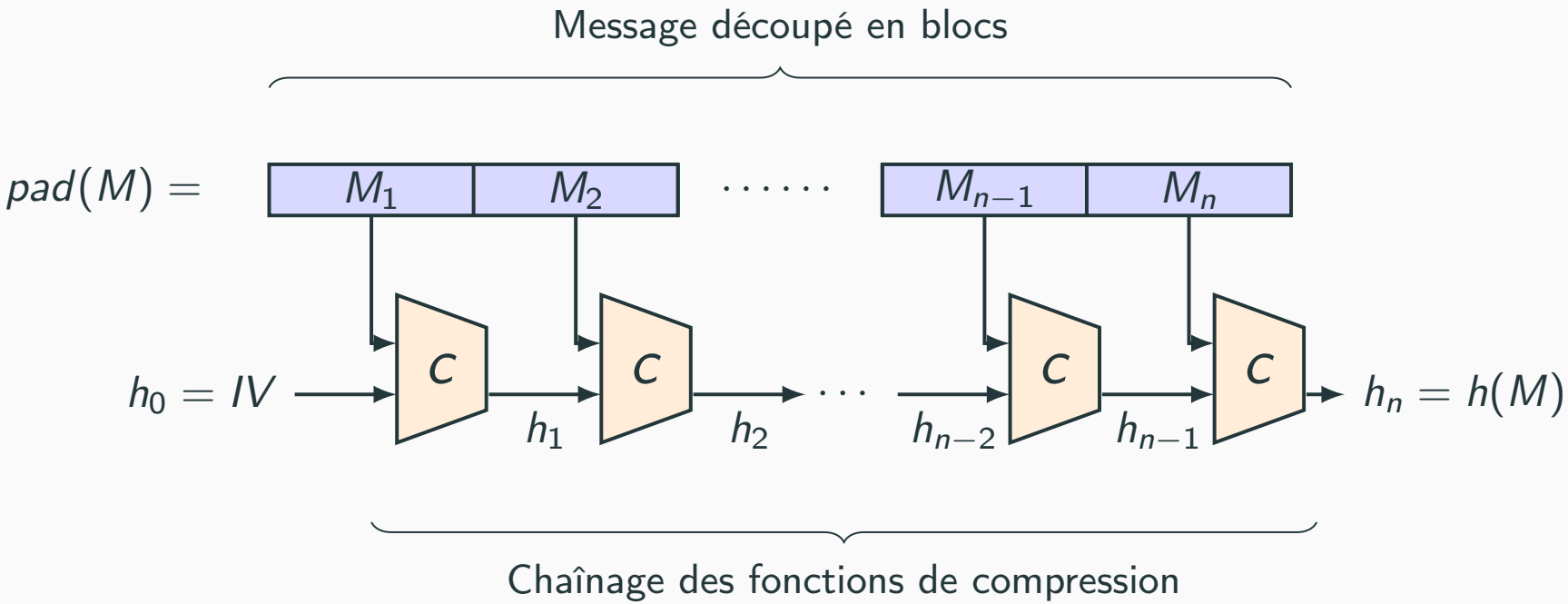
Trouver des collisions

Si la fonction de hachage a N valeurs possibles, $\sqrt{N}$ calculs suffisent en moyenne pour obtenir une collision :

$$h : \{0,1\}^* \to \{0,1\}^n$$

La recherche brutale de collisions a plus d'une chance sur 2 d'aboutir après seulement $\mathcal{O}(2^{n/2})$ essais !

Construction de Merkle-Damgård



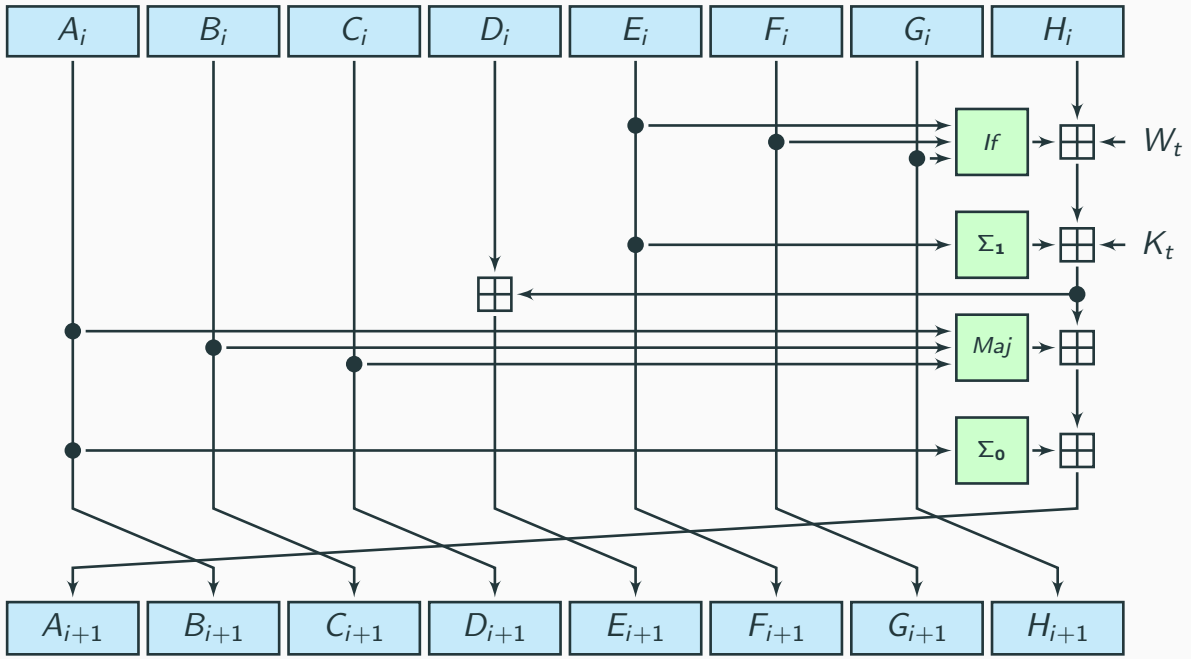
MD5, SHA-1, SHA-256 par exemple

Fonctions de hachage

SHA-256

Exemple : SHA-256

- Découpage du message en bloc de 512 bits (64 octets) (padding).
- Chaque bloc est hashé par un réseau de 64 rounds (W est calculé à partir du bloc). Le hash courant sert d'entrée pour le prochain bloc.



$$
\begin{array}{lcl}
Lf(e,f,g) & = & (e \wedge f) \oplus (\neg e \wedge g) \\
\Sigma_1(e) & = & (e >> 6) \oplus (e >> 11) \oplus (e >> 25) \\
Maj(a,b,c) & = & (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c) \\
\Sigma_0(a) & = & (a >> 2) \oplus (a >> 13) \oplus (a >> 22)
\end{array}
$$

```
$ echo -n "bonjour" | openssl dgst -sha256
SHA2-256(stdin)= 2cb4b1431b84ec15d35ed83bb927e27e8967d75f
4bcd9cc4b25c8d879ae23e18
$ echo -n "bonjour" | sha256sum
2cb4b1431b84ec15d35ed83bb927e27e8967d75f4bcd9cc4b25c8d879
ae23e18
```

http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf pour la description compléte de l'algorithme.

Fonctions de hachage

propriété d'extension de longueur

Extension de longueur

Si on connait $H(m)$, on peut calculer $H(m||padding||x)$ sans connaître m.

Les fonctions de hachage Merkle–Damgård fonctionnent par blocs :

- le message est découpé en blocs
- chaque bloc met à jour un état interne
- le hash final est simplement l'état interne final $H(m) =$ état_final

Mais cet état final est exactement l'état interne après avoir traité m.

Donc quelqu'un qui connaît ce hash peut :

- reprendre cet état
- continuer le calcul avec de nouveaux blocs.

Exemple d'attaque célèbre

Thai Duong et Juliano Rizzo.

`https://www.ioactive.com/wp-content/uploads/2012/08/flickr_api_signature_forgery.pdf`

Pour authentifier des requêtes API, flickr utilisait une signature

$$sig = SHA1(secret||parametres)$$

Par exemple,

```
method=flickr.photos.getInfo
photo_id=12345
api_key=ABC
```

Signature :

```
SHA1(secret||"method=flickr.photos.getInfo&photo_id=12345
          &api_key=ABC")
```

L'attaquant voit

```
method=flickr.photos.getInfo&photo_id=12345&api_key=ABC
api_sig = 9f4a...
```

On transforme la requête en :

```
method=flickr.photos.getInfo
photo_id=12345
api_key=ABC
&admin=true
```

On calcule, sans connaitre le secret, un hash valide

```
SHA1(secret || message || padding || "&admin=true")
```

Le serveur calcule la signature

```
SHA1(secret || requête)
```

La requête modifiée est vérifiée !

Selon les paramètres ajoutés, cela pouvait permettre :

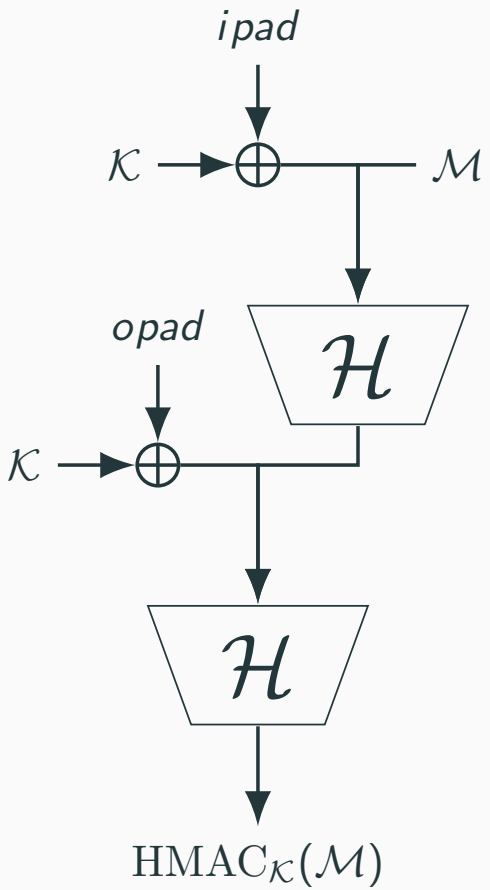
- modifier des paramètres de requêtes
- accéder à certaines opérations protégées
- contourner certaines restrictions API.

Fonctions de hachage

HMAC

Hash-based Message Authentication Code

$$HMAC_K(M) = h((K_0 \oplus opad)||h(K_0 \oplus ipad)||M))$$



But : vérifier l'intégrité de données et l'authenticité d'un message.
Utilise une clé secréte (authenticité) et une fonction de hachage (intégrité).

- h : (fonction de) hachage.
- K : clé secréte, K_0 clé secrète pré-traitée.
- m : message.
- $||$: concaténation binaire, $\oplus$: xor.

`http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf`

Chiffrement à clés publiques

Chiffrement à clés publiques

RSA

RSA

Du neuf avec du vieux. RSA est basé sur des techniques/résultats anciens :

- Nombres premiers (Euclide, antiquité)
- Petit théorème de Fermat (17ième siécle)
- Indicatrice d'Euler (18ième siècle)
- Congruence sur les entiers (Gauss, fin 18ième)

Algorithme breveté en 1977, dans le domaine public en 2000

Principe

1. Choisir deux nombres premiers distincts p et q.
2. Calculer leur produit $n = p \times q$
3. Calculer l'indicatrice d'Euler de n

$$\varphi(n) = (p-1) \times (q-1)$$

4. Choisier un entier e permier avec $\varphi(n)$.
5. Calculer d l'inverse de e modulo $\varphi(n)$.

La clé publique est le coupe (n, e). La clé privé est d. On a plus besoin de p, q et $\varphi(n)$.

Comment ça marche ?

Chiffrement/déchiffrement par RSA

M le message à envoyer. On peut le voir comme une chaine décimale.

- On découpe M en morceaux M_i de longueur fixe tels que $M_i < n$ (nombre de chiffres de M_i inférieur à celui de n)
- on chiffre chaque M_i

$$M_i \xrightarrow{E(M_i)} E(M_i) = S_i = M_i^e \mod n$$

- On envoie la suite $S_1, S_2, S_3, \ldots$
- Pour déchiffrer, le destinataire calcule $S_i^d \mod n$, qui vaut M_i.

Si une personne non autorisée intercepte et veut décrypter le message chiffré, elle doit calculer la clé privée $d = e^{-1} \mod \varphi(n)$. Il faut pour cela trouver $\varphi(n) = (p-1)(q-1)$, e étant publique. La seule façon connue de le faire est de calculer p et q, c'est-à-dire décomposer n. Or il est actuellement impossible de le faire dans un temps raisonnable si n est très grand, ce qui rend RSA sure.

Exemple : Alice voudrait que tout le monde puisse lui écrire des messages chiffrés.

- Alice choisit $p = 101$ et $q = 113$, donc $n = pq = 11413$ et $\varphi(n) = 100 \times 112 = 11200$.
- Elle choisit ensuite au hasard $e = 3533$ et vérifie par Euclide que $e \wedge \varphi(n) = 1$

r	q	u	v
11200		1	0
3533	3	0	1
601	5	.	-3
528	1	.	16
73	7	.	-19
17	4	.	149
5	3	.	-615
2	2	.	1994
1	2	u_7	-4603

Donc $11200 \wedge 3533 = 1$ et

$$1 = 11200u_7 - 3533 \times 4603$$

La dernière égalité se lit

$$3533 \times (-4603) \equiv 1 \pmod{11200}$$

d'où

$$e^{-1} \equiv -4603 \equiv 6597 \pmod{11200}$$

- La clé publique d'Alice est $(n, e) = (11413, 3533)$, et sa clé privée est $d = e^{-1} = 6597$.

Lorsque Bob veut envoyer un message $M = \ldots 1859726$ à Alice :

- il découpe en morceaux inférieurs à n : $M = \ldots 185|\underbrace{9726}_{T}$
- pour chaque morceau T, il calcule $S = T^e \mod n$:

$$S = 9726^{3533} \mod 11413 = 5761$$

- il compléte (bourrage) les valeurs obtenues par 0 à gauche jusqu'à la longueur de n, et les envoie à Alice.

$$\text{Bob} \longrightarrow \ldots|\underbrace{05761}_{S} \longrightarrow \text{Alice}$$

Lorsque Alice reçoit le cryptogramme de Bob :

- elle le découpe en morceaux S de longueur égale à celle de n ;
- elle calcule $S^d \mod n$ pour retrouver T :

$$5761^{6597} \mod 11413 = 9726$$

Un exemple (un peu) plus réaliste :

```
p=65539358441856923821403676506572753724970949277585104052373
q=90288599048695154510451472152684934674097239907586123804235 7
n=591745685626553379612634911726555458935794024190604640009363
  886664262301584649471533091171332155053082179004870120363161
e=65537
d=366395290859207922109664953887909028001824705244831861191997
  099707374915618286229486252532177398178756141604248371124017
M=12345678901234567890 // (M message à chiffer)
MC=18290367182640460300809768639810511406569855449628609985142
   770138356075414050603561430749968060769025071096549376695878 3
   //(MC message chiffré)
MD=12345678901234567890  // (MD message dechiffré)
```

Pourquoi ça marche ? cf cours d'arithmétique au S4.

Remarques

- RSA est lent.
- Difficultés à trouver des grands nombres premiers.

Clé de session

Les algorithmes asymétriques sont en général moins efficaces (en temps de calcul) que les algorithmes symétriques. On utilise le compromis de clé de session.

Clé de session

- On génére aléatoirement un clé de session (algorithme symétrique) de taille raisonnable.
- On la chiffre avec un algorithme de chiffrement à clé publique (celle du destinataire).

Chiffrement à clés publiques

Mise en oeuvre

Exponentiation rapide

RSA utilise pour chiffrer/déchiffer un calcul de puissance modulaire

$$a^c \mod n$$

L'approche brute (calculer a^c par $c-1$ multiplication) est dérisoire !

- La première simplification vient du fait que le résultat nous intéresse modulo n. On réduit donc le produit $\mod n$ à chaque étape du calcul.
- La deuxième consiste à minimiser le nombre de multiplication comme sur cet exemple :

$$\begin{aligned} a^{25} &= a * a^{24} = a * a^{12} * a^{12} \\ a^{12} &= a^6 * a^6 \\ a^6 &= a^3 * a^3 \\ a^3 &= a * a^2 = a * a * a \end{aligned}$$

Il n'a donc fallu que 6 multiplications pour calculer a^{25}.

Une version itérative dur 64 bits

```
uint64 expm(uint64 m, uint64 e, uint64 mod){
  uint128 _r=1U;
  uint128  _m=(uint128)m;
  uint128 _mod = (uint128)mod;
  while (e){
    if (e & 1) _r=(_r*_m)%_mod;
    _m=(_m*_m)%_mod;
    e>>=1;
  }
  return (uint64)_r;
}
```

- Pourquoi utilise-t-on des uint128 dans le calcul ? De telles tailles sont-elles réalistes ?
- Il faut au plus $2\log_2(e)$ mulitplication. Si $e \leq 2^{64}$, il faut au plus 128 multiplications.

Sécurité de RSA

Elle repose sur la difficulté de décomposer un entier $n = pq$ en facteurs premiers. La division brute est dérisoire, car il y a $\approx n/\ln n$ nombre premiers $\leq n$, donc $\approx 2\sqrt{n}/\ln n$ inférieurs $\sqrt{n}$.

Pour des nombres sur 2048 bits, cela représente environ 2^{1015} divisions ! En supposant qu'on dispose d'un processeur qui fasse des divisons sur 2048 bits en 1 tic d'horloge, cadencé à 4 Ghz, soit environ 2^{32} division à la seconde, il faut $2^{1015}/2^{32} = 2^{983}$ secondes. En approchant une année $365 * 24 * 3600 = 31536000$ par 2^{25}, cela fait 2^{983-25} années !

Problème : ce qui fait l'intérêt de RSA constitue aussi une limite : comment trouver des grands nombres premiers en un temps raisonnable ? Approche probabiliste.

Test de Rabbin-Miller

On rappelle que lorsque p est premier, $\mathbb{Z}/p\mathbb{Z}$ est un corps et que donc $x^2 = 1 \Leftrightarrow x = \pm 1$

Idée

Soit $p > 2$ premier. On écrit $p - 1 = 2^s d$ (d impair)

Alors d'après le théorème de Fermat, pour tout a non divisible par p

$$a^{p-1} \equiv (a^d)^{2^s} \equiv 1 \pmod{p}$$

Ainsi, en prenant successivement les racines carrées de a^{p-1}, ou bien on ne rencontre que 1, et $a^d \equiv 1 \pmod{p}$, ou bien $\exists 0 \leq r \leq s-1$ tel que $(a^d)^{2^r} \equiv -1 \pmod{p}$

Si la propriété précédente est fausse, alors p n'est pas premier, et a s'appelle un témoin.

On a le résultat suivant :

Théorème

Pour un nombre impair composé n, 3/4 au moins des entiers a, $1 < a < n$, sont des témoins de Miller pour n.

Il suffit donc de répéter le test pour suffisamment d'entiers a choisis indépendamment, pour que la probabilité qu'un entier n composé soit déclaré premier devienne très faible.

Chiffrement à clés publiques

Optimal Asymmetric Encryption Padding

OAEP

❓ Comment casser le déterminisme de certains chiffrages (comme RSA) ?

💡 Ajouter de l'aléa, et le mélanger (obscurcissement) au message.

Principe OAEP : m message, et r données aléatoires. H et G 2 fonctions de hachages.

On calcule

- $X = m \oplus G(r)$
- $Y = H(X) \oplus r$

$$OAEP(m) = X||Y$$

Très utilisé avec RSA : on parle de RSA-OAEP.

Chiffrement à clés publiques

Diffie-Hellman

Groupe multiplicatif $\mathbb{Z}_n^*$

- Pour n entier, on note

$$Z_n^*$$

  le groupe des inversibles (au sens du produit) de Z_n
- On rappelle que $k \in \mathbb{Z}_n^*$ ssi $k \wedge n = 1$ (k et n premier)

Pour $n = 8$

$$Z_8^* = \{1, 3, 5, 7\}$$

$*$	1	3	5	7
1	1	3	5	7
3	3	1	7	5
5	5	7	1	3
7	7	5	3	1

Pour $n = 9$

$$Z_9^* = \{1, 2, 4, 5, 7, 8\}$$

$*$	1	2	4	5	7	8
1	1	2	4	5	7	8
2	2	4	8	1	5	7
4	4	8	7	2	1	5
5	5	1	2	7	8	4
7	7	5	1	8	4	2
8	8	7	5	4	2	1

Groupe cyclique

- Un groupe fini G d'ordre n est dit cyclique s'il admet au moins un élément g d'ordre n
- g est appelé un générateur, et $G =< 1, g, g^2, \ldots, g^{n-1} >$

Pour $n = 8$, $Z_8^* = \{1, 3, 5, 7\}$

- $3^2 = 1$
- $5^2 = 1$
- $7^2 = 1$

Aucun générateur

Pour $n = 9$,
$Z_9^* = \{1, 2, 4, 5, 7, 8\}$
$2^2 = 4$, $2^3 = 2 * 4 = 8$,
$2^4 = 2 * 8 = 7$, $2^5 = 2 * 7 = 5$

$$Z_9^* = \{1, 2, 2^2, 2^3, 2^4, 2^5\}$$

2 est générateur

Remarque : il est inutile de tester toutes les puissances pour calculer l'ordre d'un élément, car celui-ci est nécessairement un diviseur de l'ordre du groupe.

Théorème

Lorsque p est premier :

- $Z_p^* = \{1, 2, \ldots, p-1\}$ (ordre $p-1$)
- Z_p^* est cyclique.
- Si g un générateur, tout élément x s'écrit alors sous la forme

$$x = g^a,\ 0 \leq a < p-1$$

a s'appelle le logarithme (discret) de x dans la base g

$$a = \log_g(x)$$

Pour $n = 11$, 2 est générateur

$*$	1	2	3	4	5	6	7	8	9	10	
1	1	2	3	4	5	6	7	8	9	10	2^0
2	2	4	6	8	10	1	3	5	7	9	2^1
3	3	6	9	1	4	7	10	2	5	8	2^8
4	4	8	1	5	9	2	6	10	3	7	2^2
5	5	10	4	9	3	8	2	7	1	6	2^4
6	6	1	7	2	8	3	9	4	10	5	2^9
7	7	3	10	6	2	9	5	1	8	4	2^7
8	8	5	2	10	7	4	1	9	6	3	2^3
9	9	7	5	3	1	10	8	6	4	2	2^6
10	10	9	8	7	6	5	4	3	2	1	2^5

$\log_2$ est bien défini, avec

x	1	2	3	4	5	6	7	8	9	10
$\log_2(x)$	0	1	8	2	4	9	7	3	6	5

- le calcul

$$x \longrightarrow g^x$$

  est facile (en temps). (cf algo exponentielle rapide)
- le calcul

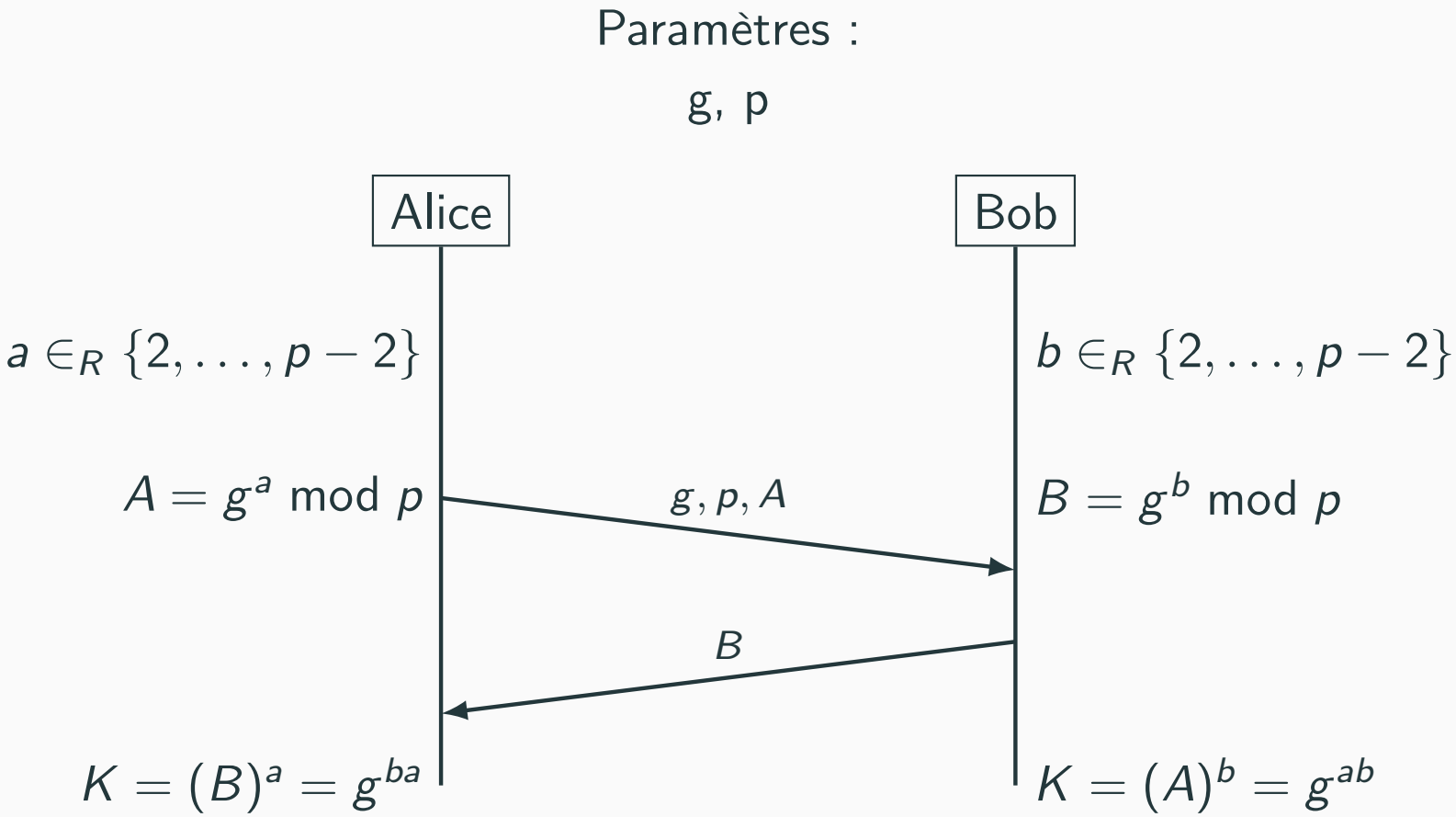
$$x \longleftarrow g^x$$

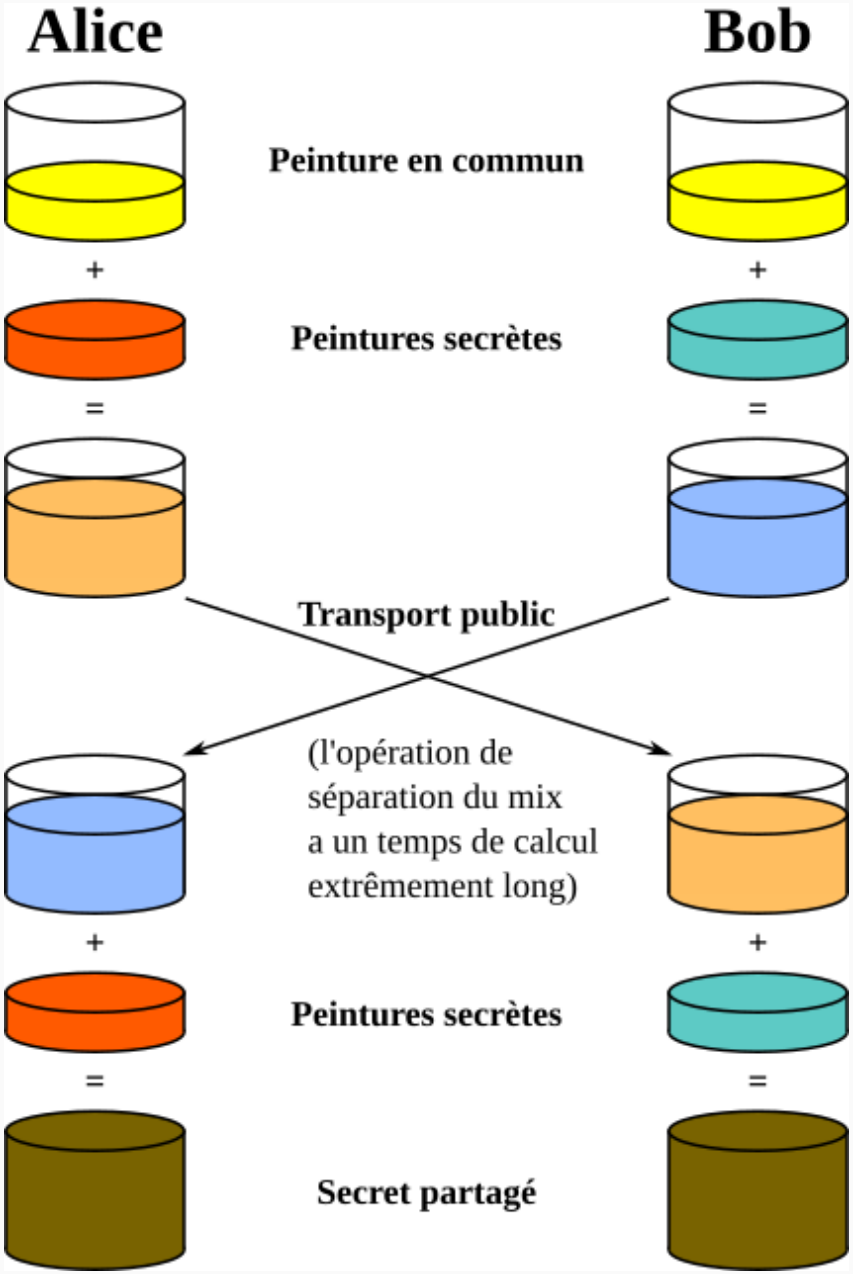
  ne l'est pas en général !

C'est cette propriété qu'utilise certains méthodes cryptographiques :

- Diffie-Hellman.
- ElGamal

Echange d'une clé avec Diffie-Hellman




Alice
Bob
Peinture en commun
+
Peintures secrètes
=
Transport public
(l'opération de
séparation du mix
a un temps de calcul
extrêmement long)
+
Peintures secrètes
=
Secret partagé

Comment trouver p, g ?

- On génère p premier, puis force brute.
  - On part de $g = 2$. On calcule g^x pour x diviseur de $p - 1$.
  - Soit g convient, sinon on passe $g + 1$.
- Nombre premier de Sophie Germain. nombre premier p tel que $q = 2p + 1$ est encore premier.
  - On évite 1 et $q - 1$ (ordre 1 et 2)
  - On part de $g = 2$. On ne calcule que g^p. Soit g convient, soit on passe à $g + 1$.

  Remarque : on peut aussi choisir g au hasard

$$g \in [2, \ldots, q - 1 = 2p]$$

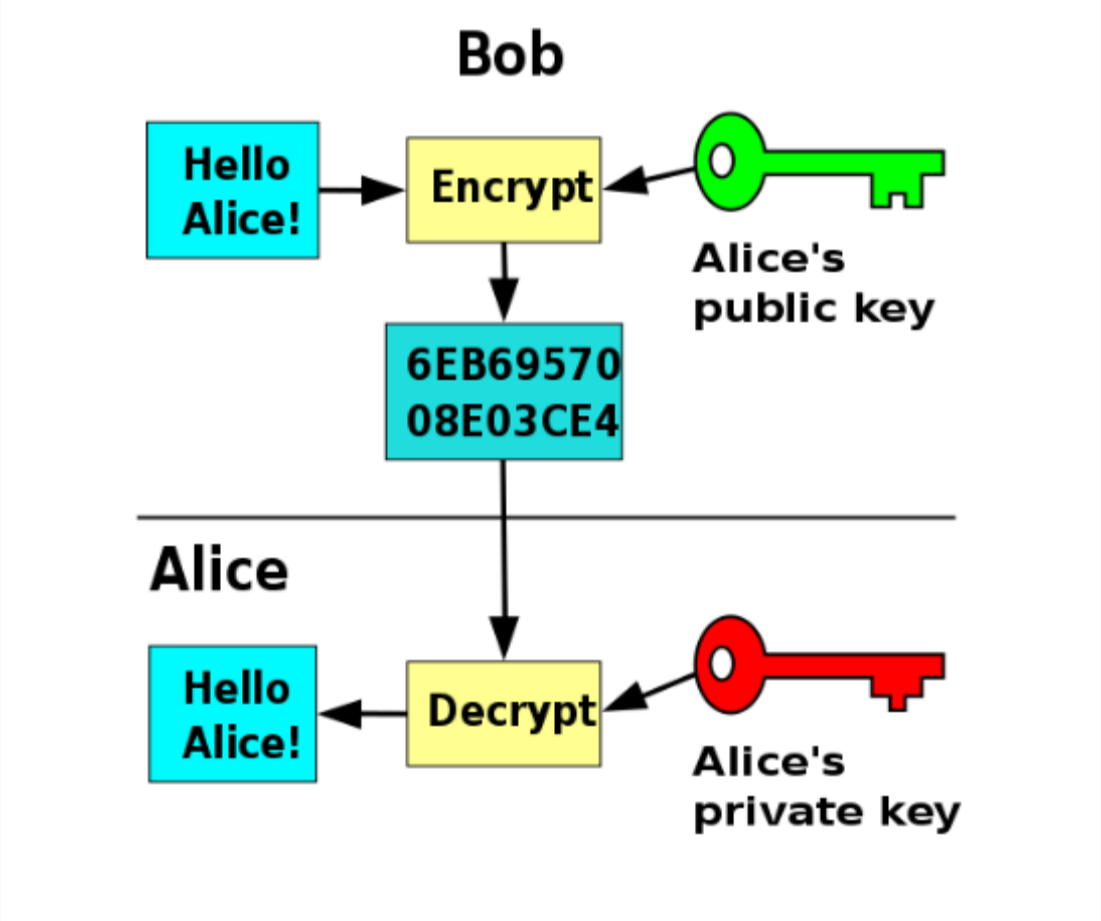
  L'ordre de g est soit p soit $2p$. Si c'est p, on reste dans le sous-groupe d'ordre p qu'il engendre.

Signatures, certificats

Signatures, certificats

Confidentialité

Rappel : chiffrement avec clé publique



- Le chiffrement asymétrique est souvent lent.
- En pratique, on l'utilise pour échanger une clé de session sysmétrique.

Signatures, certificats

Authenticité

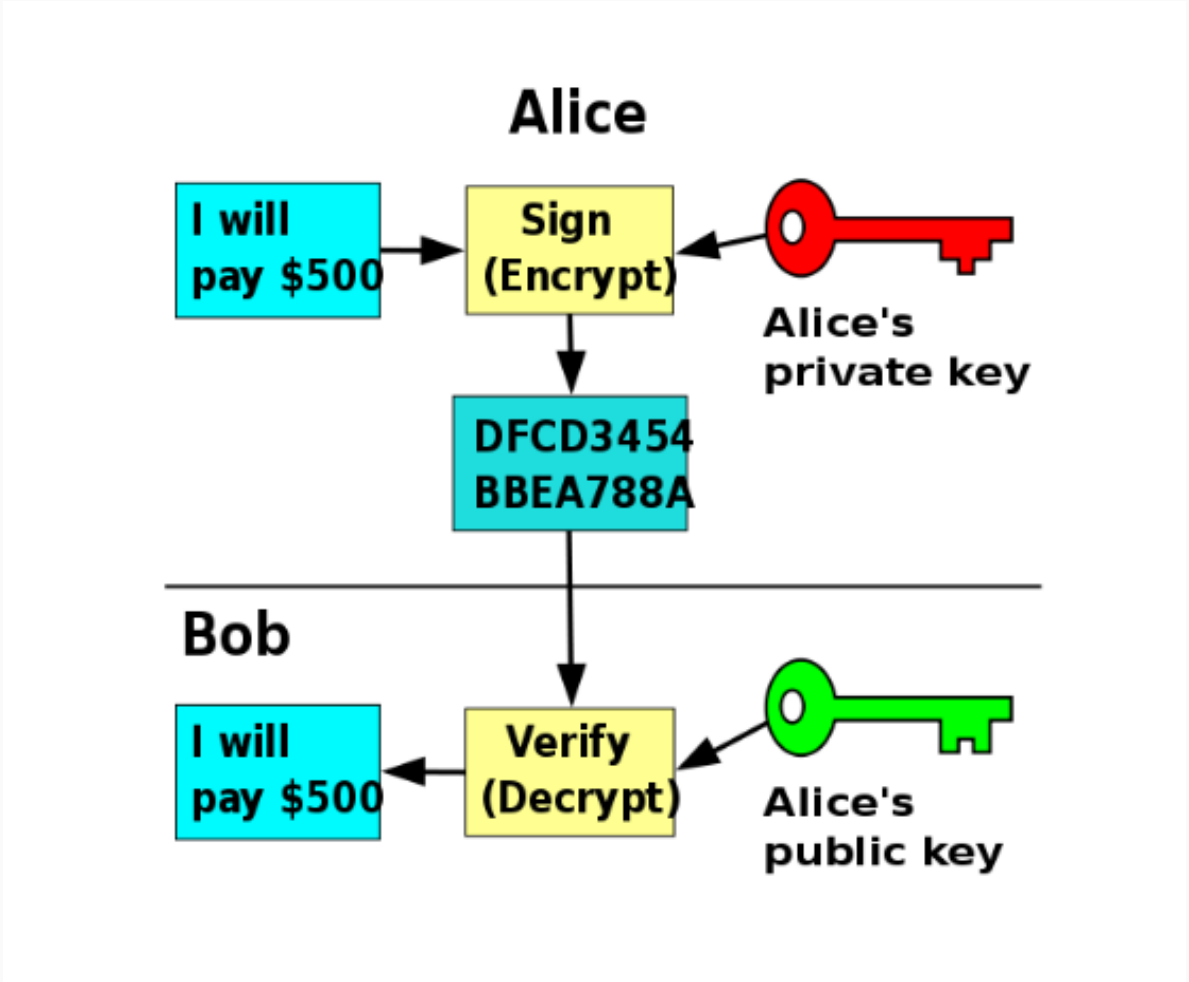
Alice envoie un message M à Bob

Bob veut être certain que le message reçu est bien d'Alice.

1. Alice chiffre le message avec sa clé privée.
2. Bob le déchiffre avec la clé publique d'Alice.

Authenticité

N'importe qui peut déchiffer le message avec la clé publique d'Alice, et vérifier que seule Alice avait pu le chiffer !


Alice
I will pay $500
Sign (Encrypt)
Alice's private key
DFCD3454 BBEA788A
Bob
I will pay $500
Verify (Decrypt)
Alice's public key

Authentification par challenge

Alice et Bob peuvent signer avec leurs clés pricées respectives.

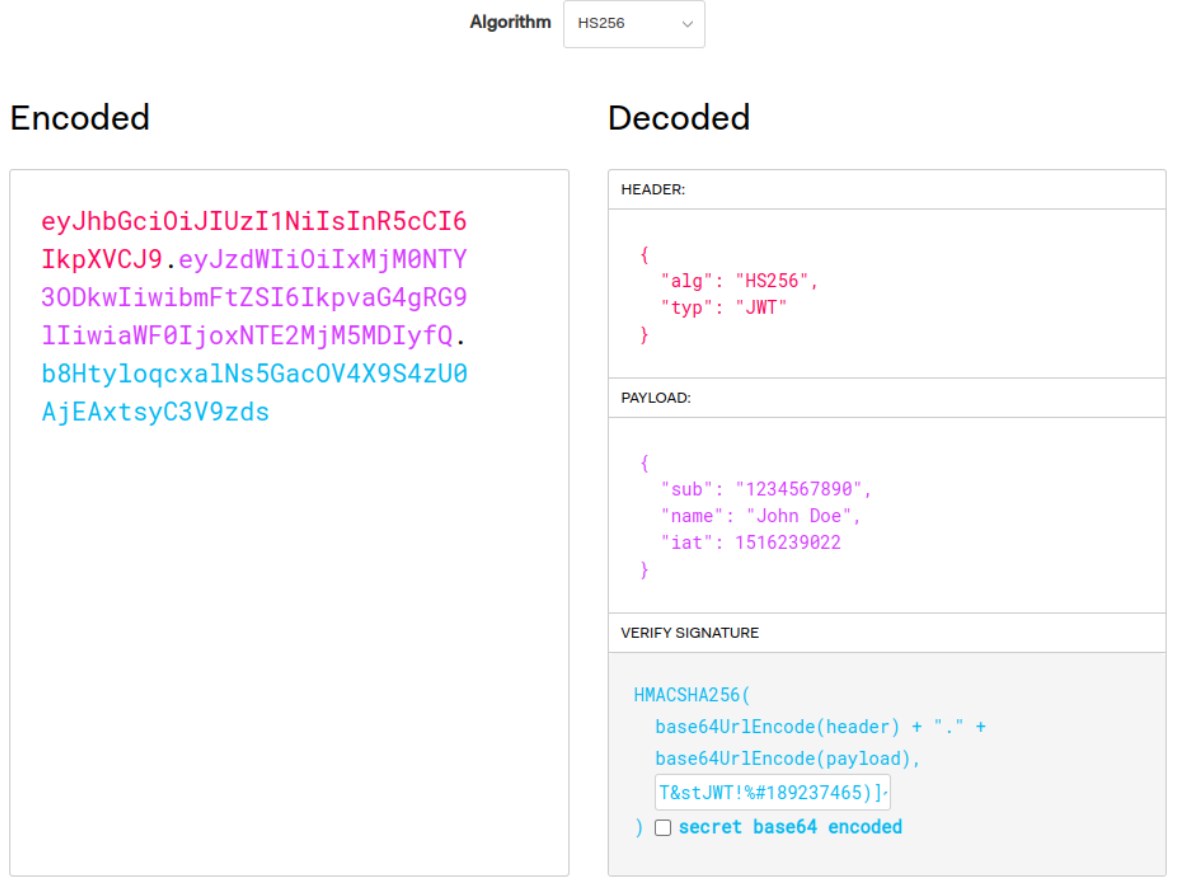
1. Bob $\rightarrow$ Alice : un nonce aléatoire unique N_B
2. Alice $\rightarrow B$: $(sign_{priv_A}(N_B), N_A)$, N_A un nonce unique.
3. Bob $\rightarrow$ Alice : $sign_{priv_B}(N_A)$
4. Alice vérifie la signature de Bob.

Il existe des algorithmes d'authenficiations mutuelles qui utilise un serveur de confiance : Needham-Schroeder (cf TD)

Utilisation avec les Json Web Token

Défini dans la rfc 7519.

- JWT permet l'échange sécurisé de jetons entre plusieurs parties,
- Vérification de l'intégrité et de l'authenticité des données (HMAC ou RSA).



Trois parties :

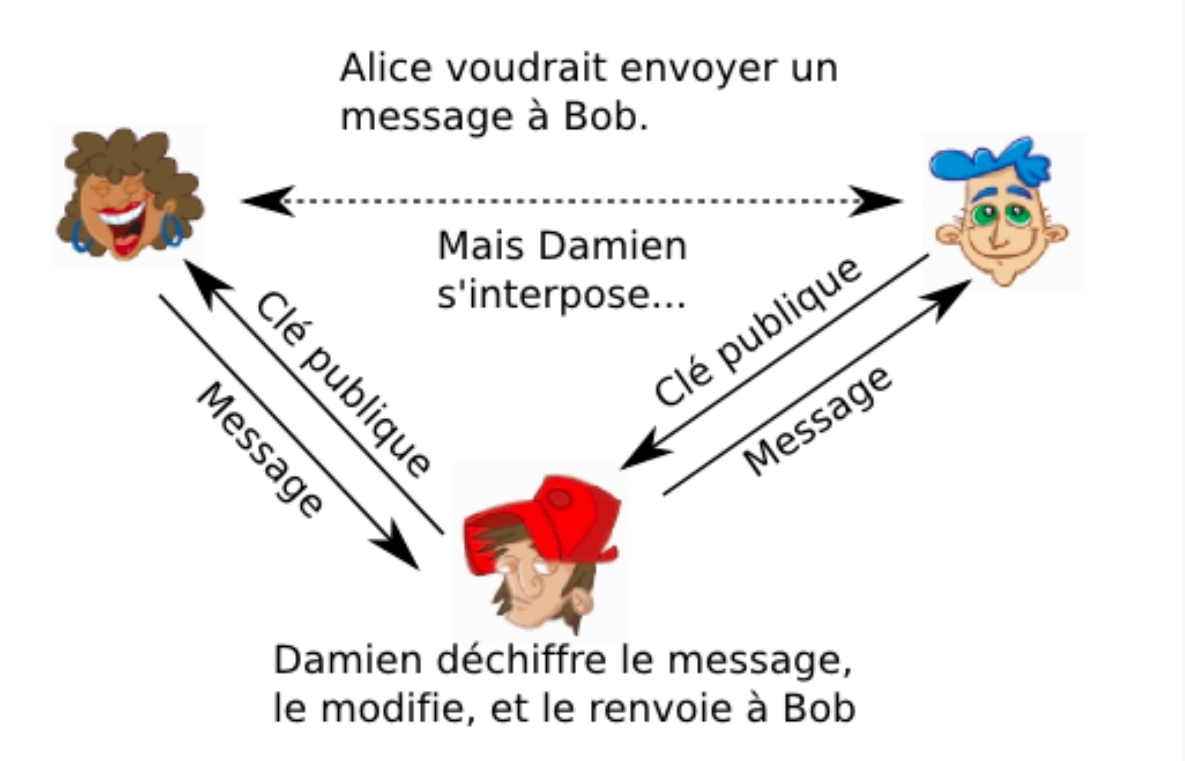
- Entête (json), qui décrit le jeton (type, algorithme),
- Charge utile (json) : user, id, date expiration, etc.
- La signature de l'entête et la charge utile encodée en base 64, concaténée avec un point. La signature est concaténée également avec un point de séparation.

L'algorithme pour la signature peut être symétrique ou asymétrique.

Quelles différences (avantages/inconvénients) par rapport à un token de sessions ?

Signatures, certificats

Public Key Infrastructure



Attaque de l'homme du milieu.

- Alice veut communiquer avec Bob
- mais Damien arrive à la fois à se faire passer pour Alice aux yeux de Bob, et à se faire passer pour Bob aux yeux d'Alice.
- Il envoie à Alice sa propre clé publique.
- Lorsqu'elle chiffre son message, elle utilise donc la clé de Damien.
- Celui peut intercepter le message, le déchiffrer avec sa clé privée, le modifier à sa guise éventuellement, puis le renvoyer à Bob en utilisant la clé publique de Bob.

Problème de l'authenticité

Comment être certain que la clé publique récupérée par Alice est bien celle de Bob, et non celle d'un imposteur ?

On peut se rencontrer :-)

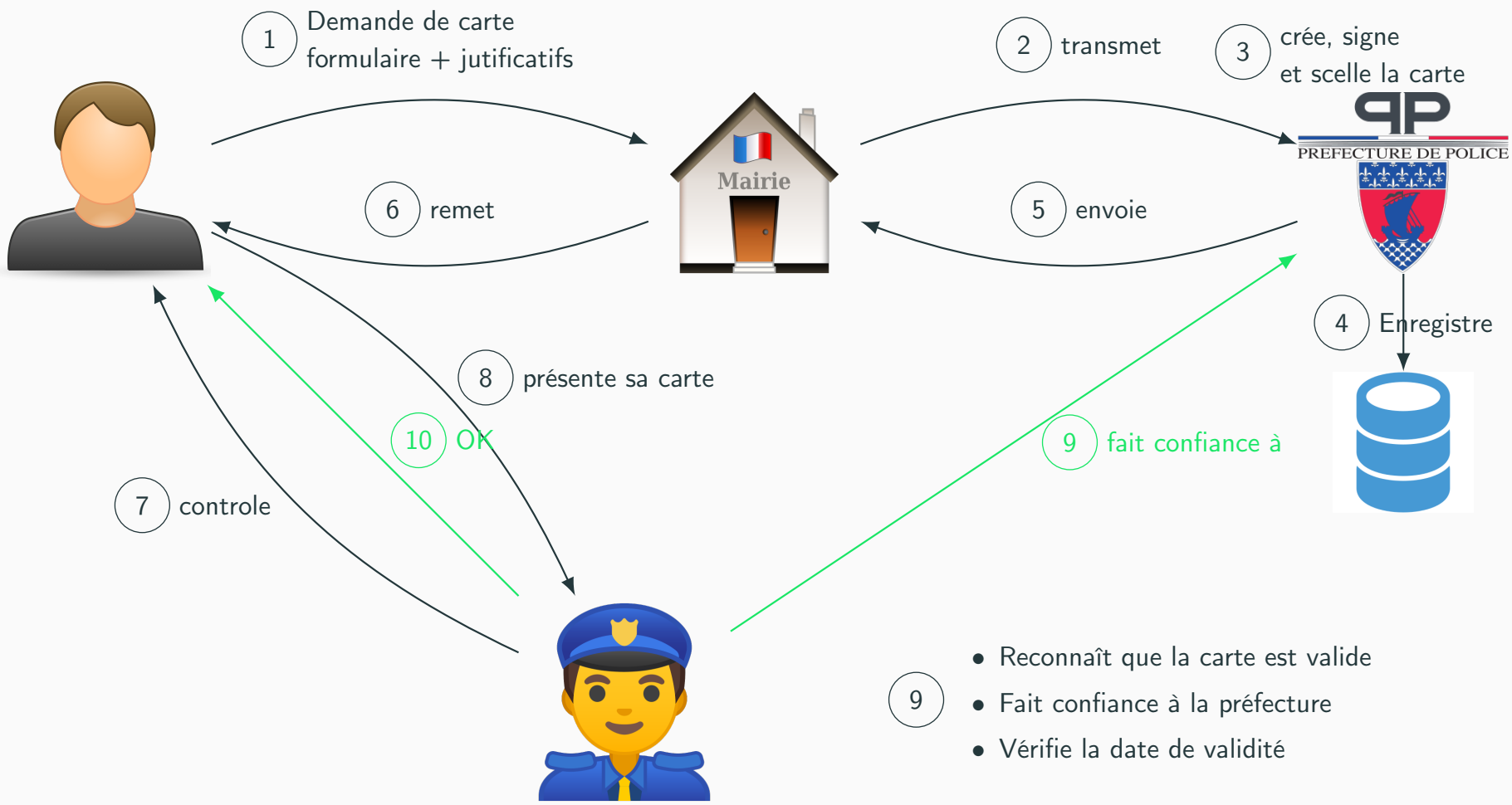
Une solution : un annuaire (un dépôt) de clés publiques

Un "organisme" maintient une liste de clés publiques (analogie avec les dns)

Utilisateur	Clé publique
Alice	12AF87BE3AB21BDA
Bob	FAB1579C2CAA6DA7
...	...
Oscar	5BDEFF128901A62D

Problème : l'organisme a aussi une clé publique. Comment garantir à nouveau son authencicité ?

Analogie avec la carte nationale d'identité



- un numéro,
- des informations sur l'identité de la personne,
- une date d'émission, et une date de fin de validité,
- des informations sur l'autorité administrative, et une signature,
- une empreinte (synthèse).

Public Key Infrastructure

PKI permet de "gèrer" les clés publiques et offre les garanties :

- confidentialité (chiffrage)
- authencité (signature)
- intégrité
- non-répudiation

Notion de certificat

Certificat de clé publique

Il contient :

- la clé publique d'une personne,
- des informations "administratives" : nom, adresse, email, etc.

Le tout est signé par une "personne de confiance".

La personne de confiance est désignée par autorité de certification (AC ou CA).

Question

Qui garantit la clé publique du CA ?

Certificat X509

Structure (`https://tools.ietf.org/html/rfc5280`)

- Version
- Numéro de série
- Algorithme de signature du certificat
- DN (Distinguished Name) du délivreur (autorité de certification)
- Validité (dates limites) : Pas avant, Pas après
- DN de l'objet du certificat
- Informations sur la clé publique
  - Algorithme de la clé publique
  - Clé publique proprement dite
- Signature des informations ci-dessus par l'autorité de certification

Le certificat de dwarves.iut-fbleau.fr

```
openssl x509 -text -noout -in dwarves.pem
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            03:8b:13:3a:2b:51:81:c4:69:d9:49:47:ed:78:28:42:ee:8
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Let's Encrypt,
                            CN = Let's Encrypt Authority X3
        Validity
            Not Before: Jan 22 09:14:43 2019 GMT
            Not After : Apr 22 09:14:43 2019 GMT
        Subject: CN = dwarves.iut-fbleau.fr
```

```
Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
       Public-Key: (2048 bit)
         Modulus:
          00:bb:0c:5a:fd:d0:a1:56:6f:3e:a9:09:50:ff:dc:
          ec:51:d7:19:3c:8e:72:88:d1:e8:9c:62:12:d3:4a:
          e0:d0:6e:8c:df:98:82:c7:61:a8:74:00:c8:e2:a1:
          d7:58:ba:bd:6d:f1:0d:a6:fb:6c:6f:e6:b7:71:48:

          ...
          9a:6b
         Exponent: 65537 (0x10001)
```

```
Signature Algorithm: sha256WithRSAEncryption
        54:22:e6:98:ec:cb:bd:4a:81:64:90:b7:7f:53:3a:29:bc:ca:
        bc:9e:d3:16:e7:8e:df:e0:dc:d8:84:f3:88:75:a7:bd:ad:cb:
        4d:20:7f:e4:de:a3:dd:7c:69:6f:fc:6b:6c:73:69:a4:14:45:
        f6:93:89:15:36:0c:69:bf:57:b7:64:a0:fc:d7:f3:4a:e9:8a:
        8c:bc:83:04:b8:52:1e:3e:5e:44:3e:85:3d:42:11:cf:4c:cc:
        6f:90:14:0b:06:3c:2a:a2:24:45:e1:8b:e6:91:a9:f2:b8:08:
        31:86:9f:41:ac:65:e5:37:fc:5c:4a:7a:c2:65:65:69:50:f6:
        9d:90:4c:9c:7b:43:0a:4f:cc:0e:8a:06:45:40:d0:b4:ad:cc:
        4c:a7:e3:14:12:eb:fd:9c:5c:dc:e6:0c:07:a3:d1:f8:29:4a:
        58:fb:43:12:e1:a5:7f:d3:b4:76:97:a0:a7:6c:ee:9a:c0:11:
        cf:2a:36:0f:52:33:24:ec:38:17:27:10:66:d1:6c:7b:dc:a7:
        07:17:27:8d:70:8b:bc:7c:c5:b2:69:be:61:2b:a3:b2:72:d4:
        f9:bc:78:8a:8c:1e:26:29:95:44:ec:9d:5d:47:ca:da:95:55:
        6f:f0:df:9e:0c:ba:90:fe:33:44:5c:1d:1c:e5:d9:4d:25:6a:
        f8:d0:82:4c
```

Vérification par le navigateur

- Vérification des dates.
- Vérification du nom du site auquel on accède.
- Vérification de la signature de l'AC :
  - utilisation de clé publique de l'ac pour déchiffrer.
  - calcul de l'empreinte
  - comparaison avec l'empreinte déchiffrée

Quand on utilise la clé publique de l'AC, on lui fait confiance (?)

IdenTrust	49.1%
Sectigo	27.1%
DigiCert Group	11.9%
GoDaddy Group	6.8%
GlobalSign	2.8%
Certum	0.8%
Actalis	0.4%
Secom Trust	0.3%
Entrust	0.3%
Let's Encrypt	0.1%

Hiérarchie des AC

- Chaque AC possède elle-même un certificat, signé par une autre AC, etc.
- $\Rightarrow$ Hiérarchie des AC.
- Le dernier certificat de la chaîne est signé par ... lui-même. On parle de certificat autosigné, ou racine.

Acteurs

Détenteur d'un certificat (Ex IUT)

- entité qui possède une clé privée
- le certificat contient la clé publique associée
- plusieurs types de certificats : serveur, vpn, etc.

Utilisateur du certificat

- récupère le certificat
- utilise la clé publique qu'il contient pour communiquer avec son détenteur.

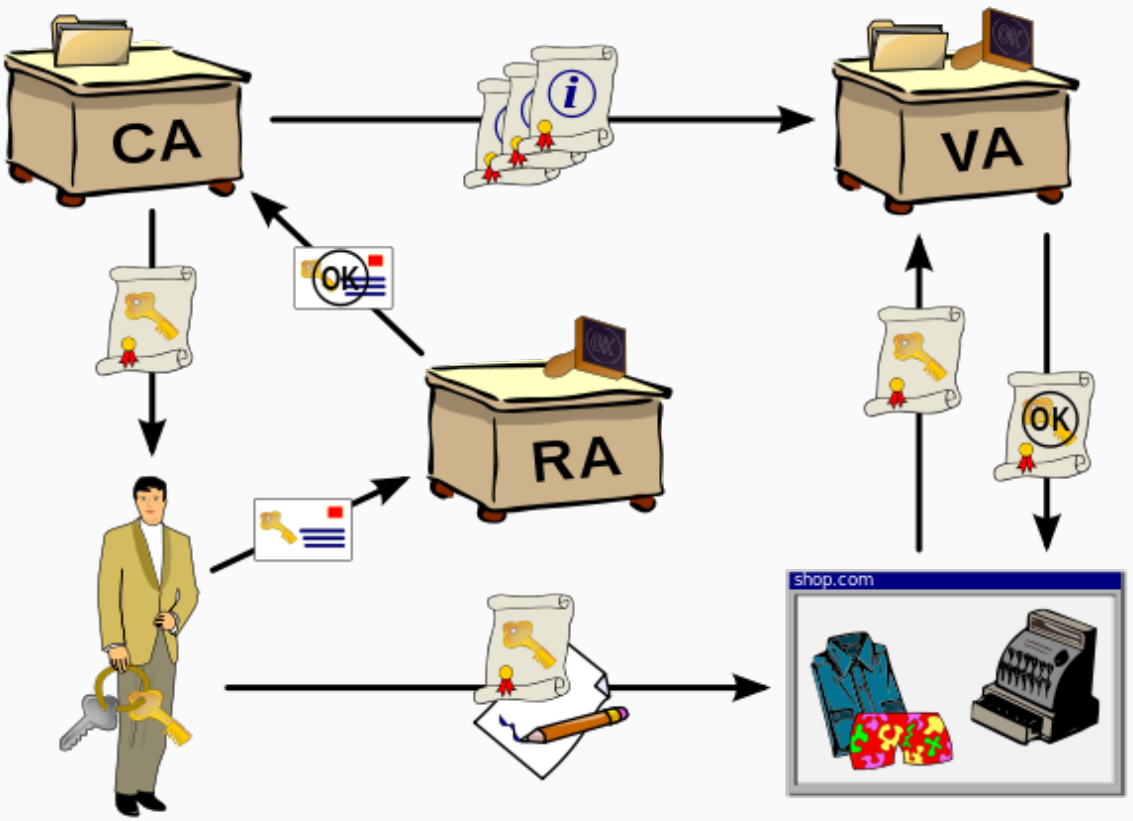
AC (autorité de certification)

- crée des ceritficats
- maintient les informations sur les crl (liste de révocation)
- publie les certificats (valides, expirés et révoqués)

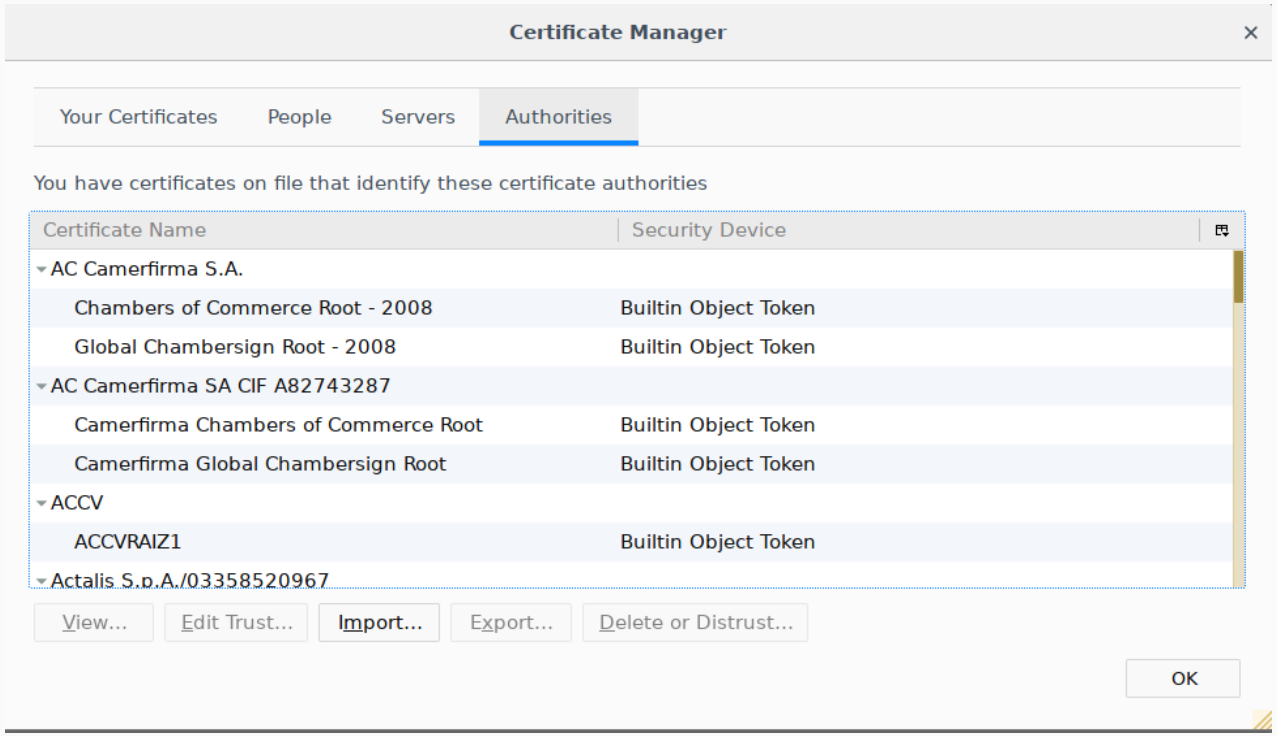
AR (autorité d'enregistrement)

- intermédiaire entre le détenteur de la clé et l'AC.
- vérifie les requêtes et les transmet à l'AC.

Emetteur de CRL (Liste de révocation)
Dépôt ou annuaire (adresse et protocole d'accès)
Archive


CA
VA
RA
OK
OK
shop.com

Certificats par défaut dans les navigateurs



Lorsque vous consultez un site web sécurisé, Firefox validera son certificat en vérifiant que le certificat signé est bien valide et en vérifiant que le certificat qui a signé le certificat parent est valide et ainsi de suite jusqu'à remonter à un certificat racine (en anglais) qui soit connu comme valide. Cette chaîne de certification est appelée la hiérarchie de certificats.

Signatures, certificats

SSL/TLS

Le protocole SSL/TLS

Initité par Netscape (années 1990), Secure Sockets Layer est devenu Transport Layer Security (TLS)

- Permet de garantir
  - l'authentification du serveur
  - la confidentialité des données échangées
  - l'intégrité et l'authentification de l'origine des données échangées
  - l'authentification du client (optionnel)
- Permet d'encapsuler de manière transparente des protocoles de la couche application
  - HTTP (80) →HTTPS (443)
  - IMAP (143) →IMAPS (993)
  - POP3 (110) →POP3S (995)
  - STARTTLS (pour IMAP, POP3, SMTP, FTP, etc.) sur le même port

Mécanismes cryptographiques dans TLS

- TLS offre le choix entre plusieurs mécanismes cryptographiques pour être capable de s'adapter aux contraintes de chaque application et de chaque système
- Authentification du serveur (et du client, optionnellement) :
  - clé publique : RSA, DSS, ECDSA
  - clé secrète ou mot de passe partagé : PSK (Pre-Shared Key), SRP (Secure Remote Password)
  - pas d'authentification : ANON
- Échange de clés :
  - clé publique (toujours la même clé privée côté serveur) : RSA
  - Diffie-Hellman statique (même problème) : DH, ECDH
  - clé secrète ou mot de passe partagé : PSK, SRP
  - Diffie-Hellman éphémère (clés privées propres à chaque connexion ; garantit la forward secrecy) : DHE, ECDHE

Mécanismes cryptographiques dans TLS

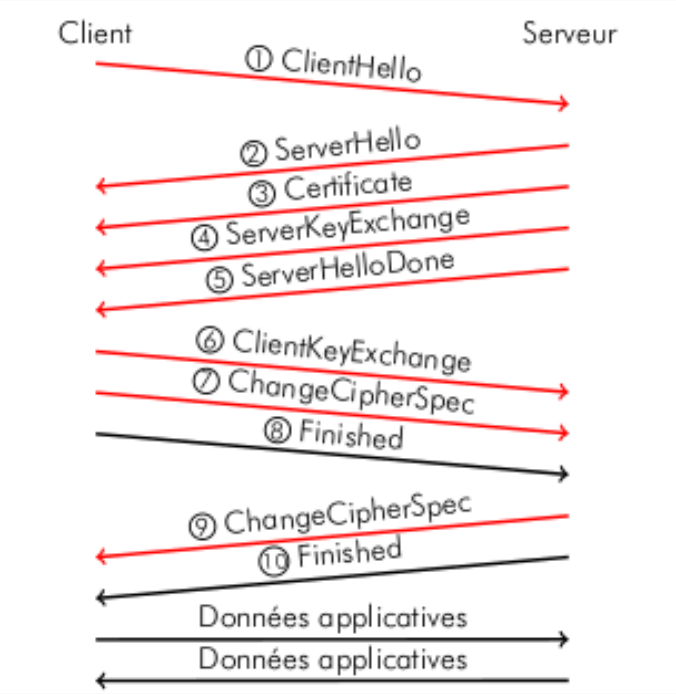
- Chiffrement :
  - chiffrement par bloc : AES-CBC, 3DES-CBC, DES-CBC, etc.
  - chiffrement par bloc avec mode authentifiant : AES-CCM, AES-GCM, etc.
  - chiffrement par flot : RC4
  - pas de chiffrement : NULL
- Intégrité et authentification de l'origine des messages :
  - HMAC : HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc.
  - mode authentifiant du chiffrement par bloc : AEAD
- Une combinaison de ces mécanismes est appelée cipher suite
  - par exemple : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- En bonus, TLS peut aussi supporter un mode de compression des données : NULL ou DEFLATE

Fonctionnement général

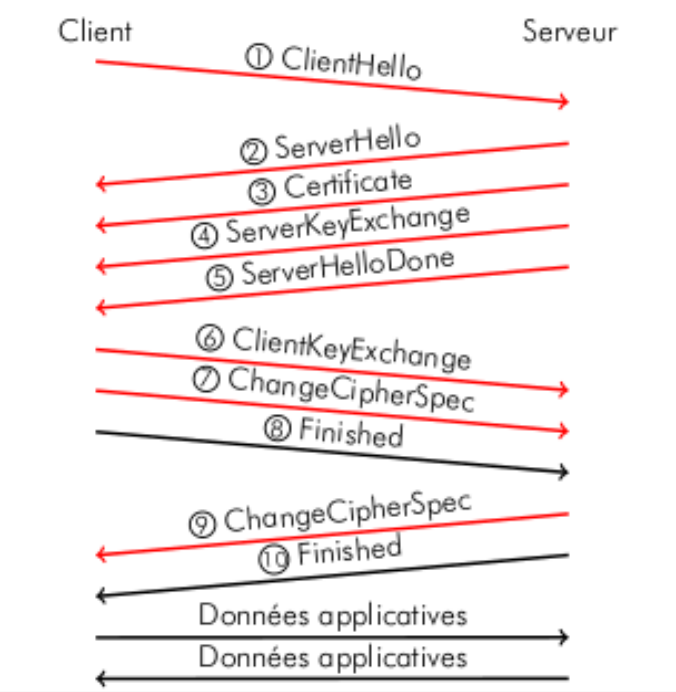
Session TLS $\Rightarrow$ 2 phases : négociation, puis communication. Le client et le serveur s'échangent :

- la version du protocol qu'ils implantent.
- les algorithmes dont ils sont capables.
- des nombres aléatoires
- les certificats (au moins pour le serveur)

Le chiffrement utilisé pour l'échange sera le plus puissant commun aux deux parties. (une des deux parties a le droit de dire non !)



1. le client envoie un `ClientHello`, contenant notamment les suites cryptographiques qu'il prend en charge ;
2. le serveur répond `ServerHello` qui contient la suite retenue ;
3. le serveur envoie un message `Certificate`, qui contient en particulier sa clé publique au sein d'un certificat numérique ;
4. le serveur transmet dans un ServerKeyExchange une valeur éphémère qu'il signe à l'aide de la clé privée associée à la clé publique précédente ;
5. le serveur dit sa mise en attente avec un `ServerHelloDone` ;



6. après validation du certificat et vérification de la signature précédente, le client poursuit l'échange de clés en choisissant à son tour une valeur éphémère et en la transmettant dans un `ClientKeyExchange` ;
7. le client signale l'adoption de la suite négociée avec un `ChangeCipherSpec` ;
8. le client envoie un Finished, premier message protégé selon la suite cryptographique avec les secrets issus de l'échange de clés éphémères précédent ;
9. le serveur signale l'adoption de la même suite avec un `ChangeCipherSpec` ;
10. le serveur envoie à son tour un `Finished`, son premier message sécurisé.

Établissement d'une connexion SSL/TLS

Contexte : un client souhaite établir une connexion SSL/TLS avec un serveur

Objectifs :

- se mettre d'accord sur une cipher suite commune
- authentifier le serveur
- échanger des clés secrètes pour le chiffrement et l'authentification des messages

Protocole de handshake en 4 étapes

Handshake SSL/TLS simple

1. Client $\rightarrow$Serveur
   - ClientHello : plus haute version du protocole supportée, liste des cipher suites et modes de compression supportés
2. Serveur $\rightarrow$Client
   - ServerHello : version du protocole, cipher suite et mode de compression choisis
   - Certificate (opt.) : la clé publique du serveur dans un certificat X.509 permettant d'en vérifier l'authenticité
   - ServerKeyExchange (opt.) : une clé publique Diffie-Hellman pour l'échange de clés
   - ServerHelloDone

Handshake SSL/TLS simple

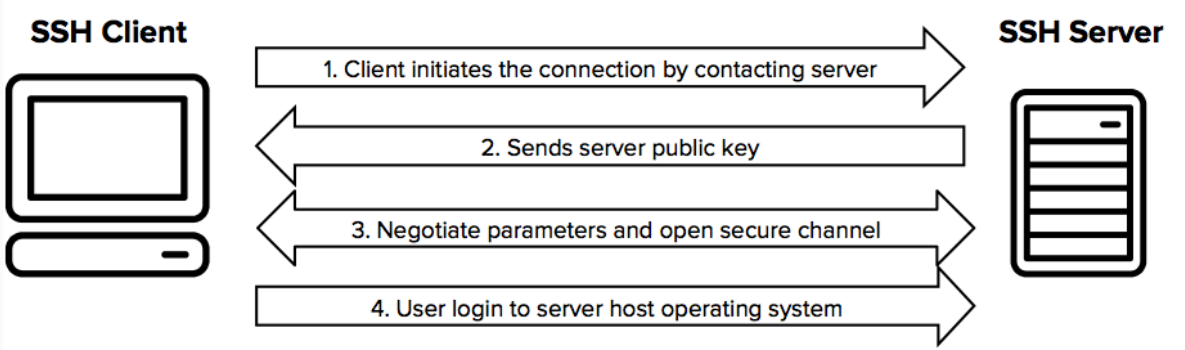
3. Client →Serveur
    - ClientKeyExchange : soit un secret (PreMasterKey) chiffré avec la clé publique du serveur, soit une clé publique Diffie-Hellman pour l'échange de clés
    - ChangeCipherSpec (marque la fin du handshake côté client)
    - Finished : message chiffré et authentifié contenant un MAC des messages précédents du handshake
4. Serveur →Client (si le Finished du client est valide)
    - ChangeCipherSpec (marque la fin du handshake côté serveur)
    - Finished : message chiffré et authentifié contenant un MAC des messages précédents du handshake
5. Si le Finished du serveur est aussi valide, la connexion est établie

Remarque : HTTPS garentit seulement que le transport est sécurisé, le reste ...

Signatures, certificats

SSH

SSH



SSH

Secure Shell (ssh) un protocole (et des programmes) de communication sécurisé. Le protocole de connexion impose un échange de clés de chiffrement en début de connexion.

Par la suite, tous les segments TCP sont chiffrés.

Plusieurs implantations.

La plus utilisée : `https://www.openssh.com/`, notamment dans les distributions GNU/Linux.

Création d'un canal sécurisé

Lors du premier échange, le client ne connaît pas le serveur. Le serveur propose alors une clé hôte qui servira par la suite au client de moyen d'identification du serveur.

```
ssh monnerat@gatekeeper.iut-fbleau.fr
The authenticity of host 'gatekeeper.iut-fbleau.fr (37.58.131.227)' can't be established.
ECDSA key fingerprint is SHA256:WpU+1HEhy+um959+ivORbsRPuDpFX1Dpn854wsnBVSs.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'gatekeeper.iut-fbleau.fr,37.58.131.227' (ECDSA)
 to the list of known hosts.
monnerat@gatekeeper.iut-fbleau.fr's password:
```

Un hachage (ecdsa-sha2-nistp256) de la clé publique du serveur a été rajouté dans le fichier `.ssh/know_hosts`

```
gatekeeper.iut-fbleau.fr,37.58.131.227 ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTY
```

- Le client échange alors avec le serveur une clé secréte (chiffrement symétrique), chiffré avec la clé publique du serveur.
- Le client envoie au serveur son identifiant et son mot de passe pour vérification.

Authentification par clé

Exemple sur le serveur git de l'iut (dwarves.iut-fbleau.fr).

Lors de l'authentification, le serveur vérifie que vous êtes bien titulaire de la clé privée correspondant à la clé publique qu'il détient, en général dans `.ssh/authorized_keys`

Génération des clés sur le client `ssh-keygen`

```
ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/denis/.ssh/id_rsa): /home/denis/.ssh/key_dem
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/denis/.ssh/key_dem.
Your public key has been saved in /home/denis/.ssh/key_dem.pub.
The key fingerprint is:
SHA256:KWBTisXrFtHNuiDM8pqiPrAirrGylXwk9d4dmMGU8pM denis@portable_denis
The key's randomart image is:
+---[RSA 2048]----+
|    ....o..      |
|   ooooo+        |
| o. *o +o.       |
|. +o++. E=       |
| o.o.oo.S..      |
|...+o..o . .     |
|o++.. . . .      |
|O= .             |
|#+.              /
+----[SHA256]-----+
```

On communique alors au serveur sa clé publique. Sa clé privée doit être conservée précieusement.

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,2E052D339B053414BBD0C3C692C176CD

ofKJty0n6eEtIv6RRwqxS6a9yqueFIRjWrECp89LfyzApeIy6gXpR/cwj9oxUwKU
5HD1Uerdw0zMjL0wT7+s4q3qeNFuF/ARNEWiRYH1APxG7alFEAVjA+hH4ychNQmP
RTA8GXVF6JkKgnmg/QoyFBQ5kKX2joWs9K1LF01IsfP4iwzPrxaHDW016pRccbGI
WcSdHNNOytwiEm8+C8fjQDY5czUr2GAdEa518B89NeqfhrE6WnFMAYA7IgdH3x3H
9y0j6CaJjljOv3sR7urSnCiROAQjzMa0q/4nlPtyk9uZ695CMKJPpCgSj1B4xBPP
elboJBHhQVgJdvQZUCLdPcZPuVcOzht7qfCJIaz4Hpe56EDhjrdiVwO8NDgoHjHc
y7Ue+4CcrAFWX1pAa4pc2VmQvjxrYuyW74+jPhMN+RMybCrHPfWjKHyJXyxMYUIE
WZiyKPCxC7dwLeQYwi3X+NhznHBKAKLHCKBbMCzsHwdQCfrviR937u/1XoW1o8E1
fo7QwEflnfA4PCDCwYBPte6QgMoN5WPaKyhdpsO77G25NvQitWyF2GjR4Nqq7AIj
uOU9SpgkesricuAKAblAqRIX1ySgCO9o8/e7vvBIbeUlGjUn5mZKtkjWjeOd/MlN
XLF4bXsvwPDZi9H6NNMgBYj0aafk2/pP+pcI79/ggezTVcjvguNmP+GPnrWiGaZh
Yy04rKW3Lw5Xz8cMGSQthq4rSQ8ohwiVNGHtEeIo1Gh8WrF4Fn1YyRLU46cMRB0Y
C60tV6NWLpJ6BnqxxQZQ3h2vvT3SikGUUdrzOT54f6LgEgPtlbEyCqpI7BmBLAgs
iiDi6x5/sQCLQ9pGMMeLtymPmvKG2coT3noHaPhJDL1EtZampHVe3guG8tNMhabg
kUezCTSGlwYWgWXG191/Znt4yUeAZGZ4+kDcFOUIMo1VtgpmLOvL6zrd7qk9eaut
YrtZVS7uYTRY4RCzDuZgxw62CArVkRMYaw1byzy9EtxBWEUQoZnBYJp12T82wSuu
G9IYELzrmC3G0lw/Zf1kb6tHbsqbngEuj044g4CL1gEznEAX+aCVz7T0P/gJpZY/
E2Oz1UpzkXJXBVYfRw4LCtQ+Iz6RSZ1kVIOfq2OIaIfClto5frYIVfsne1Mfz7BS
j7js0ico21v38tCoUaU5m4BWX63xaLHElbz5YOqhiM8I2JATRrIkTevIs3syCYQB
YEKx9NBGLuvibFiyGEPBF0IcGdFx8zMJ6MzC/T2geSJs4gm0Ot3bJiWW4Nn84RGg
mB2xMKa2hVhjFKrRtk8UDQcrSP2t8ttlQEUfU4RMEysa/lR3jawRe978AM/dVHC2
IiIYPZnndMrdIkG4REN5umdhXgDcsxv9PbdtLTn4cucNz1A62oX7Tt4hjMuncjSK
aDF+Rq8Ypjww7qYB8oV1ra9MK3z5At0l4AqJ58qrOFnNbRe6HcxCUAaPWyGNEajj
SQduRF4K4VRvA1sVuRbM+763NdPZ6JtvQDsGJozK/s1F15K4EuS6TpSwLdip9c9A
CoMBuHefY+viSrTyr7372x/nCorCVLM3xqV4/+ArgGUOmKppSMC3eVJlqd6f+yg+
-----END RSA PRIVATE KEY-----
```

Extensions ssh

- SCP
- SFTP
- Port Fowarding - Tunnel