4.4. Exponentiation modulaire

- En cryptographie asymétrique, en particulier avec l'algorithme RSA, on a souvent besoin de calculer $b^e (mod\, m)$ explicitement où $b; e; m$ sont de "grands" entiers naturels. Ce calcul s'appelle exponentiation modulaire. b est appelé la base, e l'exposant et m le module.
- Considérons le calcul de $341^{943}\ (mod\ 1403)$ .La méthode naïve consisterait à élever 341 à la puissance 943 puis effectuer la division euclidienne du résultat par 1403, le reste de cette division étant le nombre cherché.
- Une première difficulté est d'avoir à effectuer 942 multiplications (en pratique, les nombres sont beaucoup plus grands) suivies d'une division euclidienne.
- Avec cette méthode, on est conduit à manipuler des nombres de plus en plus grands.
- Il existe un algorithme efficace, appelé exponentiation par carrés (ou exponentiation rapide) qui combine deux opérations élémentaires : l'élévation au carré et la multiplication par la base b.

4.4. Exponentiation modulaire

- On commence par décomposer l'exposant e de la manière suivante : si e est pair, on le divise par 2, sinon on lui retranche 1 et on le divise par 2 et ensuite on recommence avec le quotient jusqu'à ce qu'on arrive (nécessairement) à 1. On écrit dans l'ordre inverse d'obtention cette suite de nombres. Dans notre exemple avec e = 943, cela donne la suite : 1, 2, 3, 6, 7, 14, 28, 29, 58, 116, 117, 234, 235, 470, 475, 942, 943.

- $Pour\ b = 341, on\ a\ modulo, 1403 : b^1 \equiv 341, b^2 \equiv 1235, b^3 \equiv 235, b^6 \equiv 508, b^7 \equiv 659, b^{14} \equiv 754, b^{28} \equiv 301, b^{29} \equiv 222, b^{58} \equiv 179, b^{116} \equiv 1175, b^{117} \equiv 820, b^{234} \equiv 363, b^{235} \equiv 319, b^{470} \equiv 745, b^{475} \equiv 102, b^{942} \equiv 583, b^{943} \equiv 980$

- Le calcul a nécessité seulement **16** multiplications au lieu de **942** avec la méthode naïve !

4.5. Nombres premiers

- Définition

Un nombre $p \in \mathbb{N}$ est premier s'il a exactement deux diviseurs, à savoir 1 et lui-même. L'ensemble des nombres premiers sera désigné par $\mathbb{P}$

- Remarque

0 n'est pas un nombre premier car il a une infinité de diviseurs. 1 n'est pas premier non plus car il n'a qu'un seul diviseur : lui-même. Le plus petit nombre premier est donc 2 et c'est le seul qui soit pair.

Les 25 nombres premiers $\leq$100 sont : 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 et 97.

L'importance de la notion de nombre premier vient de :

- Proposition

Tout entier $n \geq 2$ a un diviseur premier.

et surtout du résultat suivant, connu sous le nom de théorème fondamental de l'arithmétique :

4.5. Nombres premiers

- Théorème

Tout nombre entier n > 0 peut s'écrire comme produit de nombres premiers. Cette écriture est unique à permutation près des facteurs (par convention, 1 est le produit de 0 nombres premiers).

Par conséquent, si n > 1, il existe $p_1, \dots, p_k \in \mathbb{P}$*, deux à deux distincts, et tels que :*

$$n = p_1^{\alpha_1} \dots p_k^{\alpha_k} \text{ avec } \alpha_1, \dots, \alpha_k > 0$$

Ce théorème est essentiellement un théorème d'existence. Dès que l'entier n est suffisamment grand, il est impossible de trouver (avec les connaissances actuelles et le matériel existant), en un temps raisonnable, sa décomposition en facteurs premiers.

C'est sur cette impossibilité pratique que repose une importante partie de la cryptographie moderne. En particulier, c'est pour cela que les nombres premiers sont un ingrédient indispensable à la cryptographie asymétrique.

Qu'entend-on par nombre suffisamment grand ?

Le standard actuel : ***2048 bits***

5. Fonction de Hachage

5.1. Fonctions à sens unique

Généralités

Une fonction à sens unique est une fonction facile à calculer mais en revanche difficilement inversible.

Si $f: X \to Y$ est une telle fonction, alors, pour $x \in X$, $f(x)$ est facile à calculer et, pour $y \in f(x)$, il est difficile de trouver $x \in X$ tel que $y = f(x)$

Pour chaque $y \in f(x)$, dire qu'il est difficile de trouver $x \in X$ tel que $y = f(x)$ signifie qu'avec les moyens et connaissances actuels, il est impossible de trouver un tel x en un temps raisonnable sauf à compter sur une chance, elle, tout à fait déraisonnable.

Ces fonctions à sens unique ont été introduites au milieu des années 1970 et sont un des piliers de la cryptographie moderne.

5.1. Fonctions à sens unique

- Quelques applications
  - Stockage des mots de passe
  - Le protocole de ***Diffie-Hellman*** qui permet l'établissement d'un secret partagé entre deux entités à travers un canal non sécurisé
  - Lors de l'échange de données entre deux entités à travers un canal non sécurisé, les fonctions à sens unique permettent d'authentifier les parties en présence et de garantir l'intégrité des données transmises.

5.2. Définitions

- Une fonction de hachage (ou de condensation) est une fonction à sens unique qui prend en entrée des données de **longueur quelconque** (x)et rend une **valeur de taille fixe** ($f(x)$)
- Cette valeur est l'empreinte (ou aussi : condensé, condensat, haché, hash) du texte x.
- Une fonction de hachage est dite cryptographique si :
  - Il est très difficile de trouver le contenu du message à partir de sa signature
  - A partir d'un message donné, de sa signature et du code source de la fonction de hachage, il est très difficile de générer un autre message qui donne la même signature
  - Il est très difficile de trouver deux messages aléatoires qui donnent la même signature.
- C'est ce que l'on appelle une collision.
- Il existe nécessairement x, x' tels que $x \neq x'$ et $f(x) = f(x')$

5.3. Taille des empreintes

- Soit H de longueurs n bits. On sait que 1 bit a 2 valeurs possibles, 0 ou 1. On travaille donc sur un ensemble de 2^n empreintes possibles.
- **Paradoxe des anniversaires**
  - combien doit-on réunir de personnes afin d'avoir plus de 50% de chances d'avoir au moins deux personnes nées le même jour de l'année ?
  - La réponse, très contre-intuitive, est 23. Si on réunit 50 personnes, la probabilité est de 97,04% et, pour 80 personnes, elle est de 99,99%, une quasi-certitude.**,**
- On sait également qu'il $2^{\frac{n}{2}}$ essais pour trouver une collision au hasard. C'est ce que l'on appelle **la force d'une fonction de hachage**.

5.4. Les fonctions de hachage les plus connues

- **MD5**

Empreinte de 128 bits. On sait depuis 2004 qu'il est peu sûr. Il est recommandé de ne plus l'utiliser.

- **SHA1**

Empreinte de 160 bits. Encore très utilisé. Néanmoins, en 2005, Wang et al. ont annoncé qu'ils pouvaient produire une collision en environ 2^63 opérations, ce qui est à la limite du faisable. Aujourd'hui, on table plutôt sur environ 2^51 opérations. C'est donc une faiblesse toute relative d'autant que les textes provoquant ces collisions sont sans signification.

- **SHA2**

Empreinte de 256, 384 ou 512 bits. Cette famille de fonctions de hachage a été introduite en vue du remplacement de SHA-1. Pour l'instant aucune attaque significative n'existe.

- **SHA3**

Empreinte de 256, 384, 512 ou arbitraire bits. Les fonctions de hachage précédentes, étant construites à partir des mêmes heuristiques, sont sensibles aux mêmes attaques. Il a donc été décidé (NIST) de fournir une alternative fondée sur des principes complètement différents. C'est SHA-3 normalisée en 2015.

5.4. Les fonctions de hachage les plus connues

- Comparatif

Fonction	Taille (bits)	Taille (chars)	Performance (MB/sec)	Nombre d'opérations
MD-5	128	32	?	2^{64}
SHA-1	160	40	?	2^{80}
SHA-256	256	64	?	2^{128}
SHA-512	512	128	?	2^{256}

- Exercice

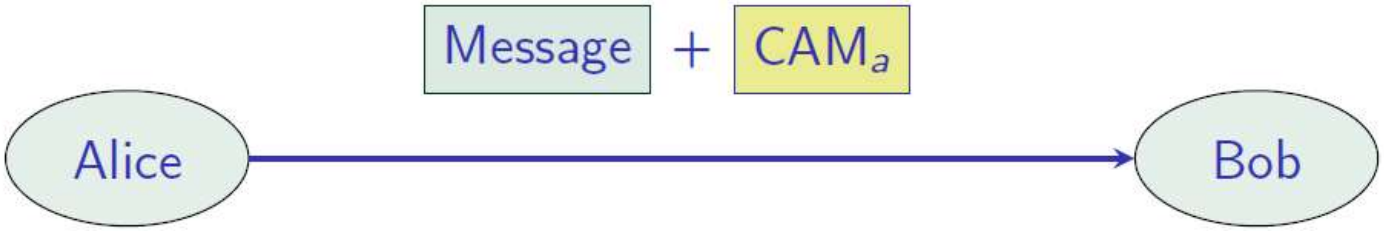
1) Générer 1 Million d'empreinte avec les algorithmes MD5, SHA-1, SHA-256, SHA-512 des variables de type string de longueur 36,49,72,85 caractères, et calculer le temps de génération des empreintes.
2) Récapituler les résultats dans un tableau.
3) Conclure

5.5. Code d'authentification de message

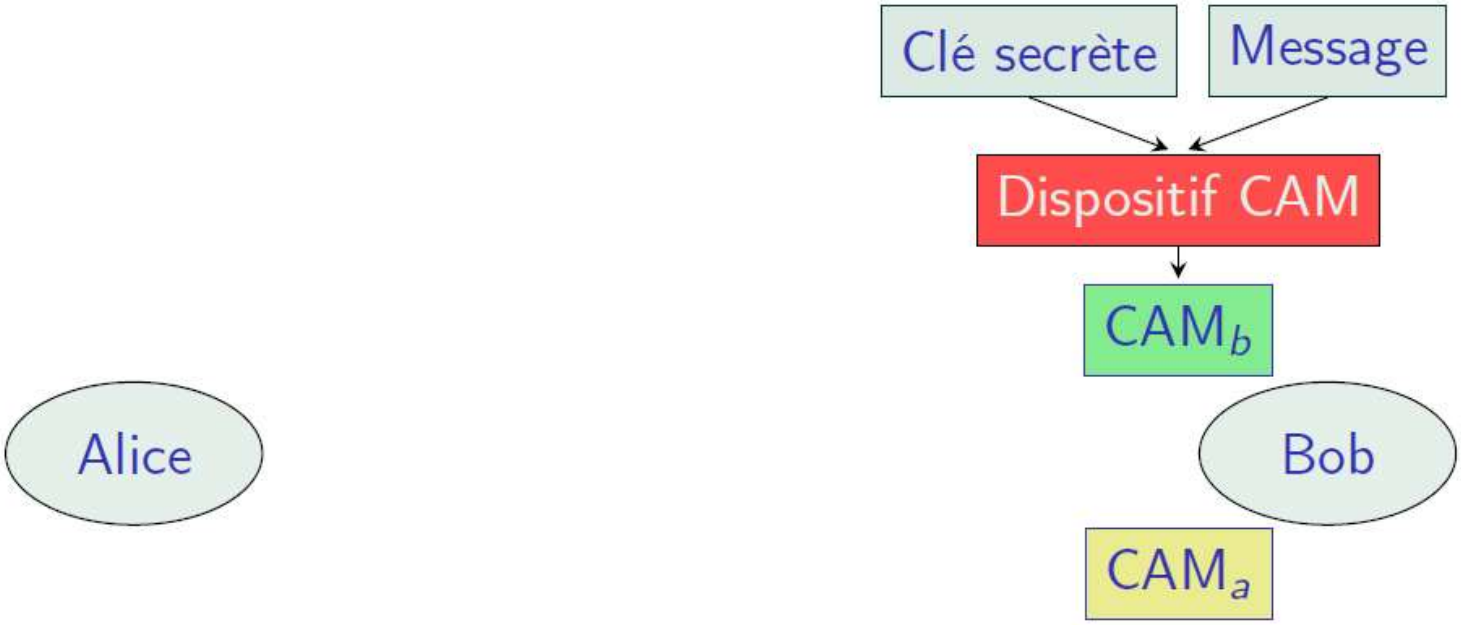
- Un code d'authentification de message (en abrégé CAM ou MAC - Message Authentication Code) est un dispositif permettant de garantir l'intégrité des données transmises à travers un canal non sécurisé entre deux entités
- Ce dispositif permet également au destinataire d'authentifier l'expéditeur des données.
- Un MAC repose sur l'utilisation d'un secret partagé entre les parties en présence et, le plus souvent, sur une fonction à sens unique du type fonction de hachage comme MD5 ou SHA-1.
- En résumé, un MAC est très similaire à une fonction de hachage hormis l'usage d'une clé secrète

5.5. Code d'authentification de message

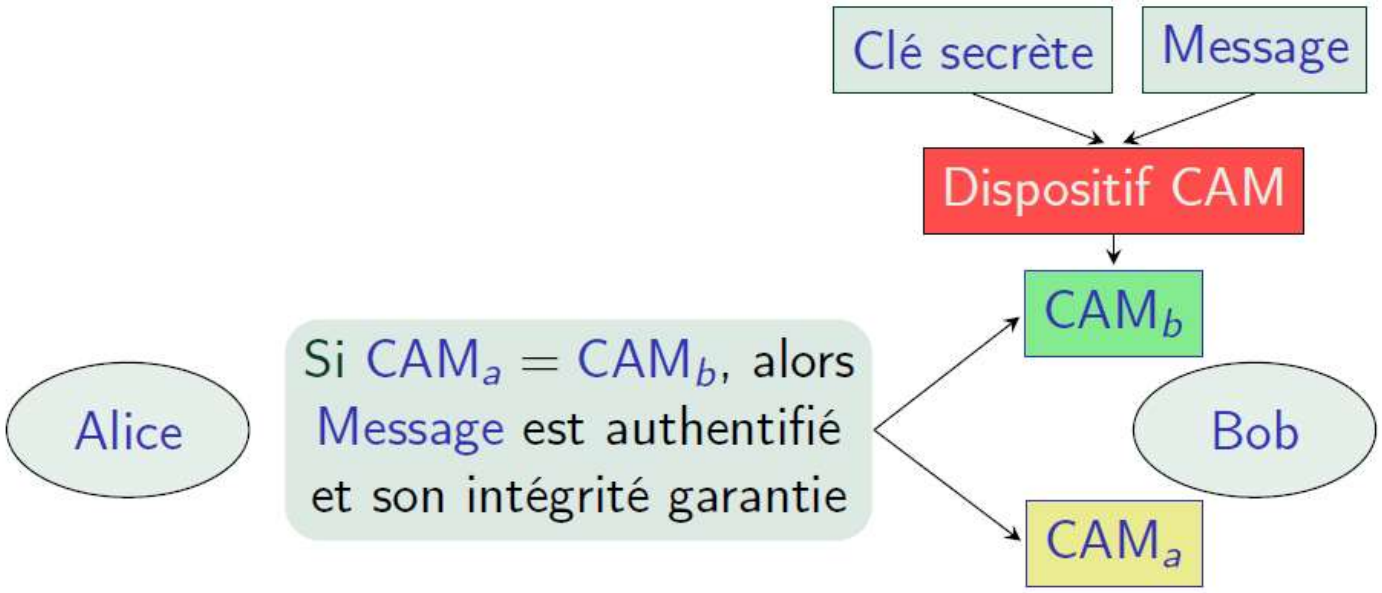
5.5. Code d’authentification de message



5.5. Code d'authentification de message



5.5. Code d'authentification de message



HMAC est le code d'authentification de message le plus utilisé à l'heure actuelle

6. Chiffrement asymétrique

6.1. Chiffrement RSA

- Système de chiffrement à clé publique
- Le système RSA, du nom de ses concepteurs *Rivest, Shamir et Adleman* est le premier système de chiffrement à clé publique robuste à avoir été inventé (en 1977).
- Il utilise une paire de clés, une clé publique pour le chiffrement et une clé privée pour le déchiffrement
- Permet également de signer des données
- La robustesse du système RSA repose sur le fait que l'on ne sait pas, avec les moyens et savoirs actuels, obtenir la clé privée à partir de la simple connaissance de la clé publique.
- Souvent en pratique, on ne l'utilise que pour transmettre la clé secrète d'un chiffrement symétrique.

6.1. Chiffrement RSA

- Concrètement supposons qu'Alice veuille échanger des données de manière sécurisée avec Bob. Elle va ainsi procéder :

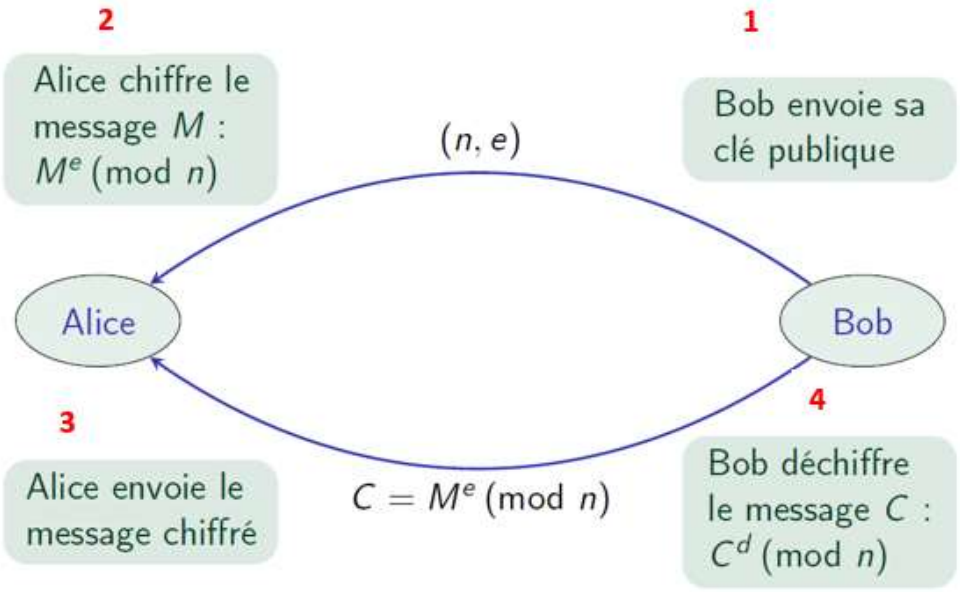
1. Elle va tout d'abord générer une clé secrète pour le chiffrement symétrique dont le choix (public) a été fait au préalable avec son correspondant.
2. Elle chiffre cette clé secrète avec la clé publique (RSA) de Bob et transmet la clé ainsi chiffrée à ce dernier.
3. Bob déchiffre la clé secrète avec sa clé privée (RSA).
4. Les deux correspondants étant maintenant en possession de la clé secrète, le chiffrement symétrique des données peut commencer.

6.1. Chiffrement RSA

- Principes de fonctionnement
- La **clé privée** est constituée de deux “grands” nombres premiers p et q
- La **clé publique** est constituée du produit $n = pq$ et d’un entier e inversible modulo $\varphi(n)$ c’est à dire un entier e tel qu’il existe un entier d avec $ed \equiv 1(mod(\varphi(n))$.
- Le **chiffré d’un message** représenté par un entier M modulo n est $M^e(mod\ n)$
- Pour **déchiffrer le message** chiffré C , il suffit de calculer $C^d(mod\ n)$ où d est l’inverse de e modulo $\varphi(n)$

6.1. Chiffrement RSA

- Schéma de fonctionnement



La recommandation actuelle pour la taille de la clé publique n est 2048 bits

6.1. Chiffrement RSA

- Exemple de génération de clés RSA avec *OpenSSL*

Pour générer une clé RSA de 2048 bits avec $e = F_4 = 2^{2^4} + 1 = 65537$ (e est le 5ème nombre de *Fermat*) :

```
$ openssl genrsa -F4 -out rsa.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................+++ # Rabin-Miller
.....................................................+++
e is 65537 (0x10001)
```

Le fichier généré *rsa.key* est au format *base64*. Attention : la clé privée est stockée en clair dans le fichier. Donc il vaut mieux procéder ainsi :

```
$ openssl genrsa -aes128 -F4 -out rsa.key 2048
Generating RSA private key, 2048 bit long modulus
.....................................+++
.....................................................+++
e is 65537 (0x10001)
Enter pass phrase for rsa.key: # MDP chiffrement AES-128
Verifying - Enter pass phrase for rsa.key: # confirmation
```

6.1. Chiffrement RSA

- Exemple de génération de clés RSA avec *OpenSSL*

Pour voir en clair le fichier rsa.key :

```
$ openssl rsa -in rsa.key -noout -text
Private-Key: (2048 bit)
modulus: # clé publique n = pq
    00:f2:22:be:06:40:b9:55:5a:a6:b3:52:8c:72:8a:
    ..............................................
publicExponent: 65537 (0x10001) # clé publique e = F4
privateExponent: # clé privée d, ed ≡ 1 (mod φ(n))
    57:a9:aa:60:7b:28:5e:35:8e:aa:d7:95:0f:96:dc:
    ..............................................
prime1: # p pseudo-premier (3 tests Rabin-Miller)
    00:fb:26:f8:56:9b:fa:05:0e:16:7e:78:d8:70:1b:
    ..............................................
prime2: # q pseudo-premier (3 tests Rabin-Miller)
    00:f6:cf:37:ef:61:ed:3a:e7:86:04:b4:1e:1f:e0:
    ..............................................
```

6.1. Chiffrement RSA

- Exemple de génération de clés RSA avec *PHP*

```php
// Create the private and public key
$config = array(
    "private_key_bits" => 2048,
    "private_key_type" => OPENSSL_KEYTYPE_RSA,
    );

$res = openssl_pkey_new($config);

// Extract the private key from $res to $privKey
openssl_pkey_export($res, $privKey);

// Extract the public key from $res to $pubKey
$pubKey = openssl_pkey_get_details($res);
$pubKey = $pubKey["key"];

$data = 'plaintext data to be encrypted';

// Encrypt the data to $encrypted using the public key
openssl_public_encrypt($data, $encrypted, $pubKey);

echo 'ciphertext is : '.base64_encode($encrypted).'<br><br>';

// Decrypt the data using the private key and store the results in $decrypted
openssl_private_decrypt($encrypted, $decrypted, $privKey);

echo 'plaintext is : '.$decrypted.'<br>';
```

6.1. Chiffrement RSA

- Robustesse et attaques de RSA

La solidité de RSA repose sur l'impossibilité, avec les moyens et connaissances actuels, de factoriser, en un temps raisonnable, $n = pq$ lorsque n est suffisamment grand (2048 bits par exemple)

Il ne faut bien évidemment qu'aucun des facteurs p et q ne soit trop petit. Il faut donc que p et q soient d'une taille analogue sans pour autant être trop proches.

- Quelques attaques

Si on choisit l'exposant public e trop petit (pour accélérer le chiffrement), par exemple $e = 3$, une attaque possible est l'attaque due à Håstad en 1985

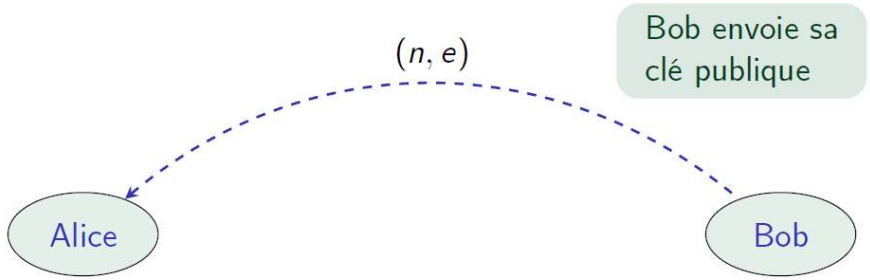
De même, si l'exposant privé d est choisi trop petit (pour accélérer le déchiffrement), *Wiener* a montré en 1989 l'existence d'un algorithme en temps polynomial permettant de retrouver d

6.1. Chiffrement RSA

Attaque “man in the middle”

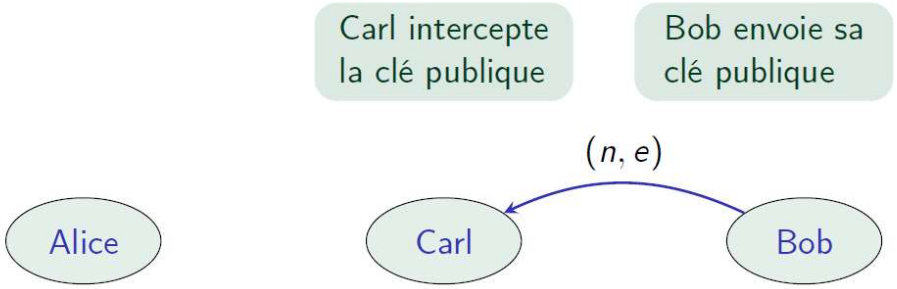
6.1. Chiffrement RSA

Attaque “man in the middle”



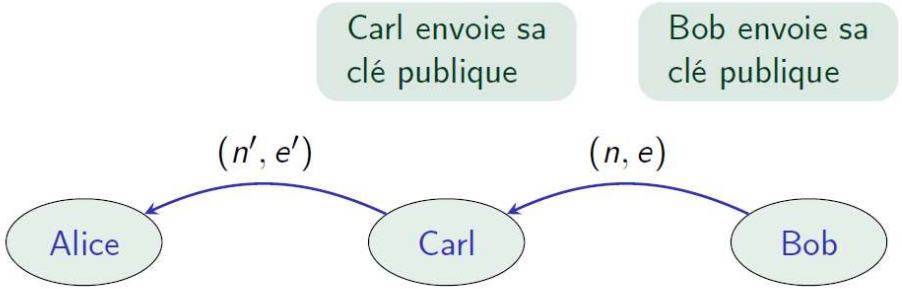
6.1. Chiffrement RSA

Attaque “man in the middle”



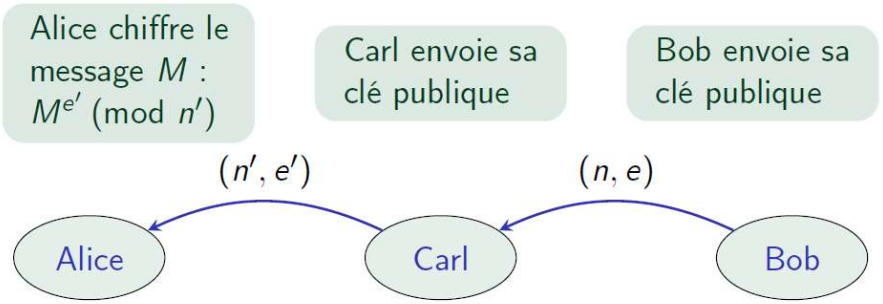
6.1. Chiffrement RSA

Attaque “man in the middle”



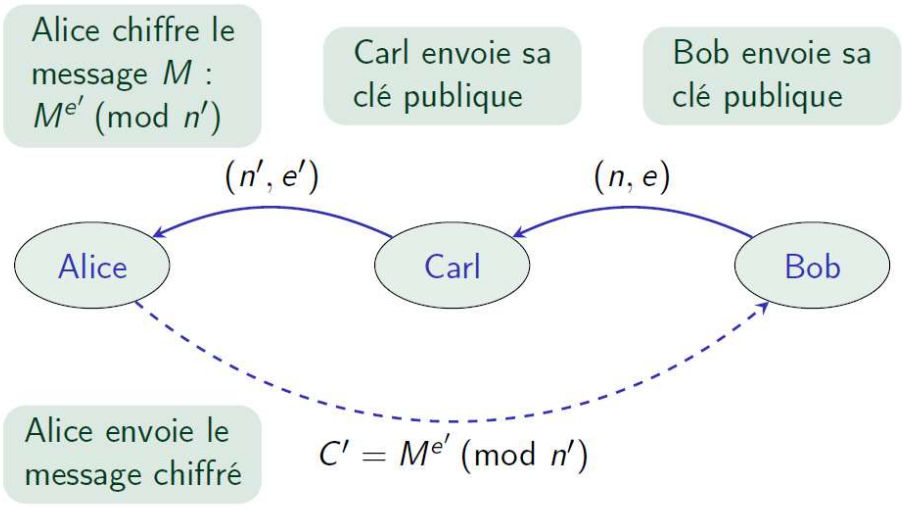
6.1. Chiffrement RSA

Attaque “man in the middle”



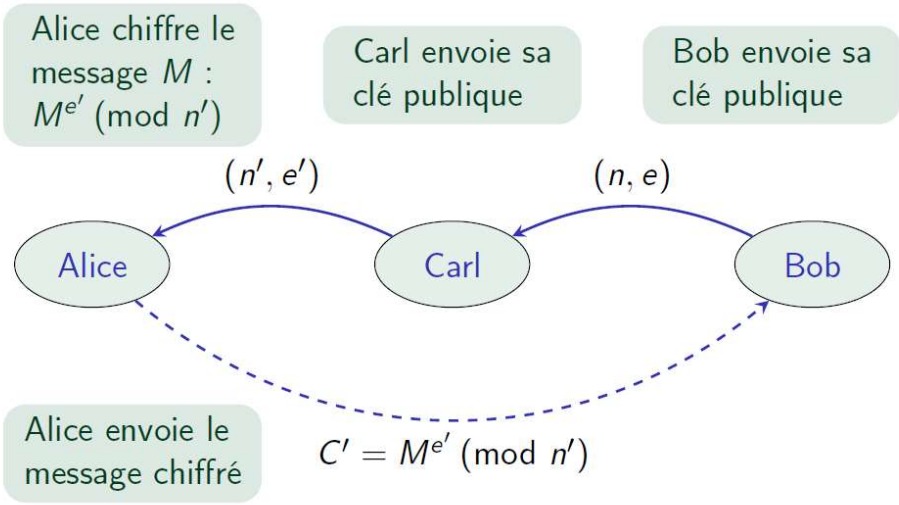
6.1. Chiffrement RSA

Attaque “man in the middle”



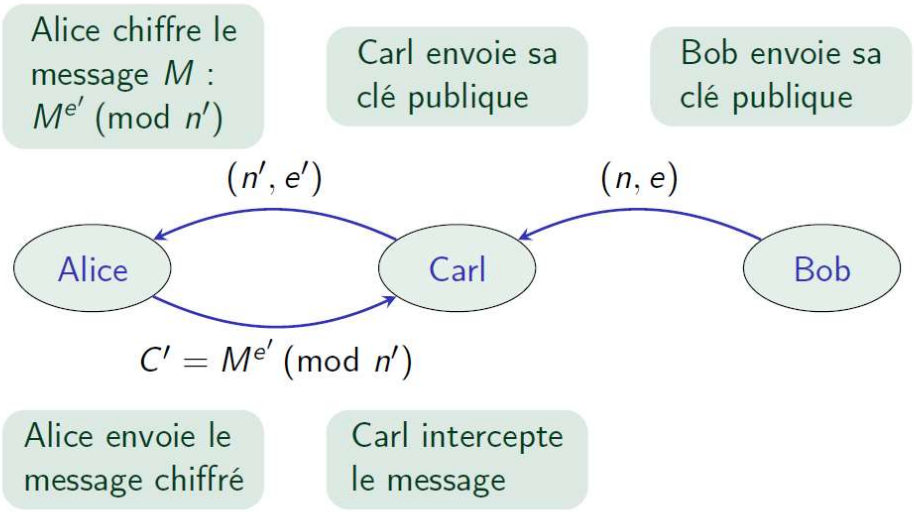
6.1. Chiffrement RSA

Attaque “man in the middle”



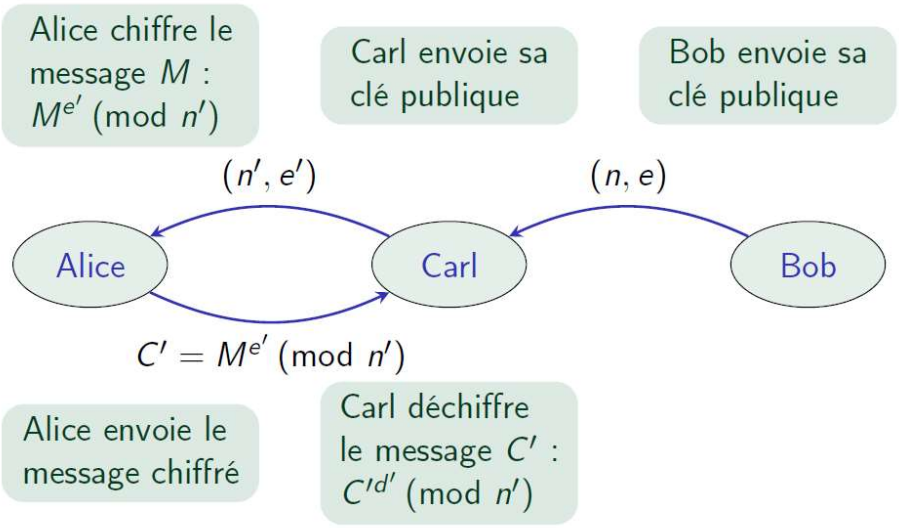
6.1. Chiffrement RSA

Attaque “man in the middle”



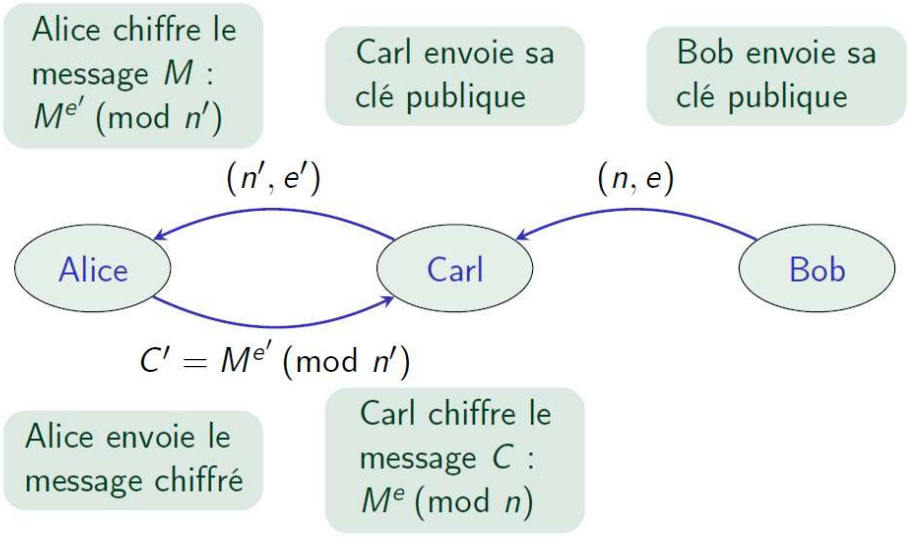
6.1. Chiffrement RSA

Attaque “man in the middle”



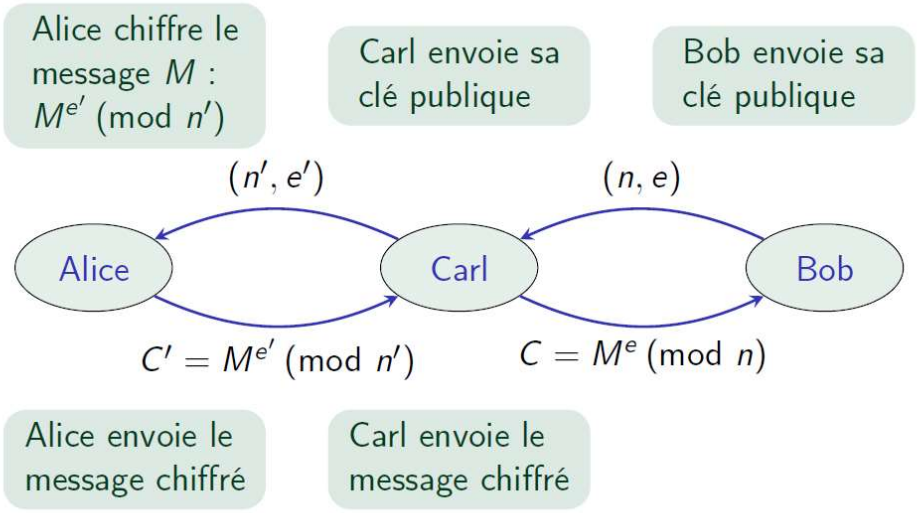
6.1. Chiffrement RSA

Attaque “man in the middle”



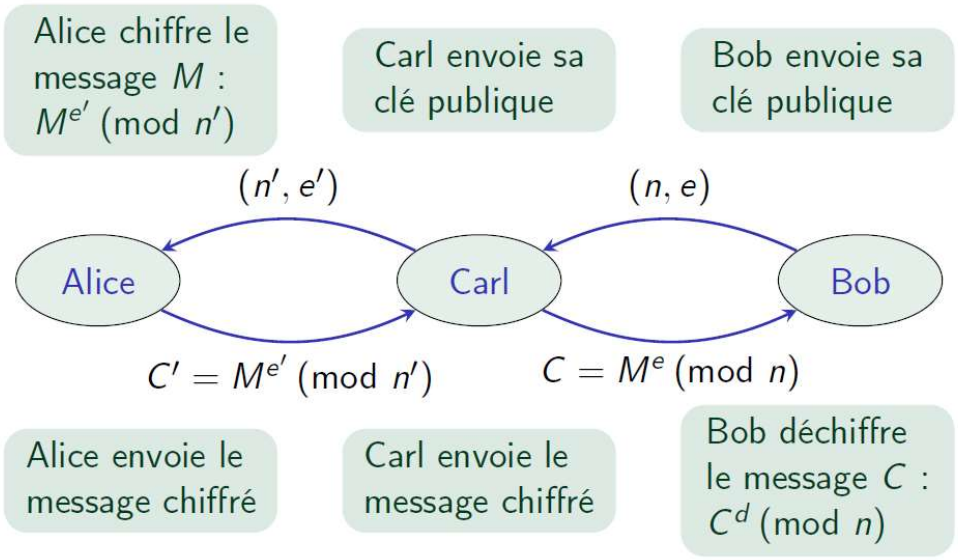
6.1. Chiffrement RSA

Attaque “man in the middle”



6.1. Chiffrement RSA

Attaque “man in the middle”



6.1. Chiffrement RSA

- Traitement du message

On a considéré que le message M était un simple entier. Bien évidemment, ce n'est pas réaliste en pratique

Le message M est en réalité une suite d'octets : un texte codé en ASCII ou en UTF-8, un paquet réseau, ...

On pourrait avoir la tentation de chiffrer le message octet par octet (et le déchiffrer pareillement). Ce serait une très mauvaise idée car un attaquant n'aurait à effectuer au plus que 256 tests pour retrouver M. Il faut donc que le message M soit suffisamment long.

6.1. Chiffrement RSA

- Le protocole OAEP

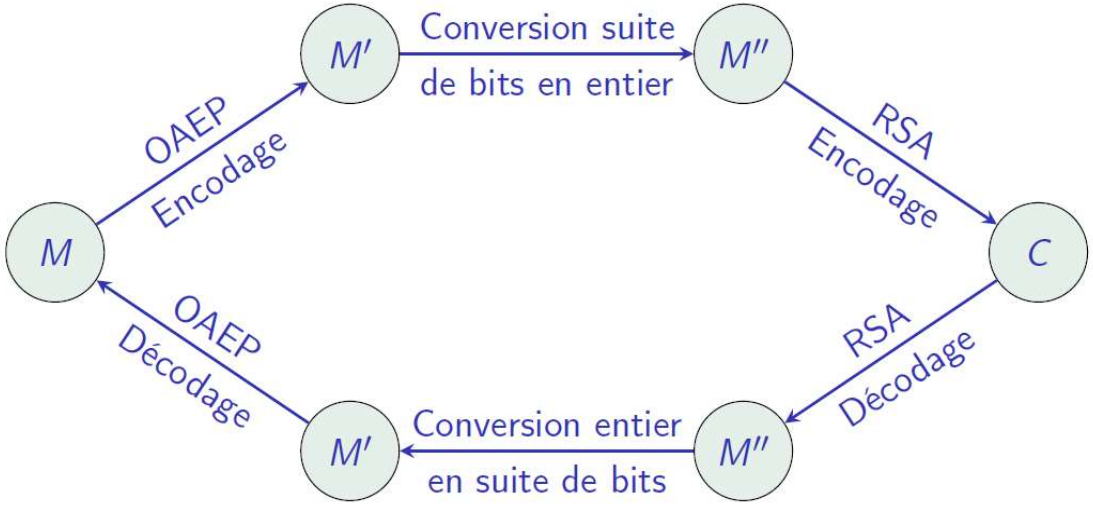
Pour parer les attaques (et d'autres également comme l'attaque par chronométrage) vues précédemment, un schéma de remplissage incluant des données aléatoires est nécessaire. Pour le système RSA, le plus utilisé est le protocole OAEP (Optimal Asymmetric Encryption Padding) créé par Bellare et Rogaway en 1994.

- Les ingrédients de OAEP sont :

1) un entier $0 < k_0 < k$, où k est la longueur en bit de la clé publique n du système RSA, tel que 2^{k_0} soit suffisamment grand
2) deux fonctions de hachage cryptographique G et H dont les hachés ont pour longueurs en bit respectives $k - k_0$et k_0,
3) un générateur aléatoire pour engendrer des suites de bits de longueur k_0

6.1. Chiffrement RSA

- Le protocole OAEP

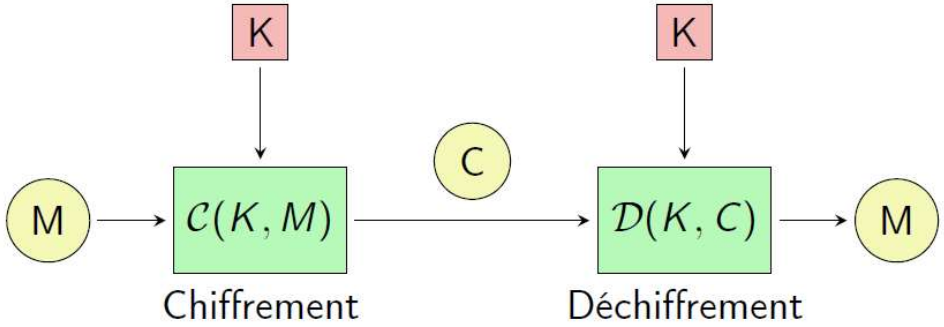


7. Chiffrement symétrique

7.1. Principe

- Un chiffré (*cipher*) est défini sur $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ est une pair d'algorithme efficients ***(E,D)*** où

$E\colon \mathcal{K} \times M \to \mathcal{C}$, $D\colon \mathcal{K} \times \mathcal{C} \to M$, tels que $\forall m \in M, k \in \mathcal{K} : D\big(k, E(k,m)\big) = m$



7.1. Principe

1. Chiffrement rapide, voire très rapide en implantation matérielle.
2. Clés plutôt courtes : 128 - 256 bits (à comparer avec RSA : 1024 -2048 bits).
3. Inconvénient majeur : partage d'un secret (la clé commune K), ce qui est toujours délicat à gérer.

2 Propriétés (Shannon) que devrait satisfaire un bon algorithme de chiffrement : **la diffusion et la confusion**.

- La propriété de diffusion signifie que des changements minimes dans les données en entrée se traduisent par des changements **importants** dans les données en sortie.
- La propriété de confusion mesure la **complexité** de l'interdépendance entre la clé, le clair et le chiffré. Plus cette complexité est grande, meilleur est l'algorithme.

7.1. Principe

- Deux types de chiffrement symétrique
- Les **chiffrements à flot**
  - on génère à partir de la clé une suite chiffrante pseudo-aléatoire de même longueur que les données. On combine cette suite, par exemple avec un XOR bit-à-bit, avec les données à chiffer.
  - RC4 (Rivest, 1987) : SSL/TLS, WEP, WPA, WPA2,...
  - E0 : Bluetooth
  - A5 : GSM
- Les **chiffrements par bloc**
  - les données à chiffrer sont découpées en blocs de taille fixe (typiquement 64 ou 128 bits). Les blocs sont chiffrés séparément et ensuite combinés selon un mode opératoire (ECB, CBC, CTR...).
  - DES (IBM et NSA, 1975) - blocs 64 bits, clés 56 bits
  - IDEA (Lai-Massey, 1992) - blocs 64 bits, clés 128 bits
  - AES/Rijndael (Daemen-Rijmen, 1998) - blocs/clés 128, 192, 256 bits
  - et aussi : RC5, RC6, Camellia,...

7.2. Chiffrement à flots

- Chiffrement de Vernam (1917)

L'espace des messages $\mathcal{M}$ est le même que l'espace de texte chiffré $\mathcal{C}$, qui est l'ensemble des chaines binaires de n bits.

$$\mathcal{M} = \mathcal{C} = \{0,1\}^n$$

L'espace des clés $\mathcal{K}$ est le même que l'espace des messages $\mathcal{M}$, ou qui est encore l'ensemble des chaines binaires de n bits

$$\mathcal{K} = \{0,1\}^n$$

La clé k est une chaine binaire de même longueur que le message.

Le chiffré est le résultat du chiffrement d'un message avec une clé particulière, ou simplement le XOR des 2. (Rappel : XOR c'est une addition modulo 2)

$$c \coloneqq E(k,m) = k \oplus m$$

7.2. Chiffrement à flots

- Exemple

On additionne les bits un à un, et on ob

$$\begin{array}{r} m: 0110111\P \\ \oplus\P \\ k: 1011001\P \\ \hline c: 1101110\P \end{array}$$

Maintenant on veut déchiffrer le chiffré

$$D(k, c) = k \oplus c$$

Comment s'assurer que le déchiffrement donnera le message en texte clair ?

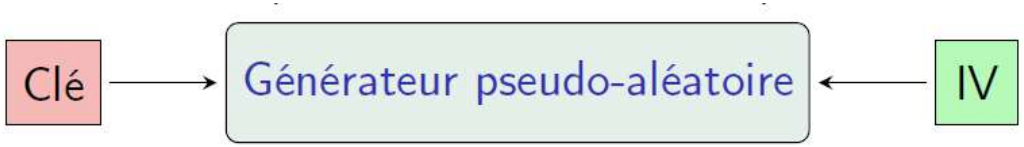
$$D\big(k, E(k, m)\big) = D(k, k \oplus m) = k \oplus (k \oplus m) = (k \oplus k) \oplus m = 0 \oplus m = m$$

- Si on a message donné m et son chiffré c, peut-on retrouver la clé à partir de m et c ?

Oui, en fait, la clé se calcule facilement par k=m⊕c

7.2. Chiffrement à flots

- Sécurité
- Un bon algorithme de chiffrement repose sur le fait que le chiffré $\boldsymbol{c}$ ne doit pas révéler d'information sur le message $\boldsymbol{m}$. **(Shannon 1949)**
- Une première difficulté : si on chiffre plusieurs messages avec la même clé, on génère le même aléa. On ajoute donc une entrée auxiliaire, le vecteur d'initialisation (Initialization Vector - IV).

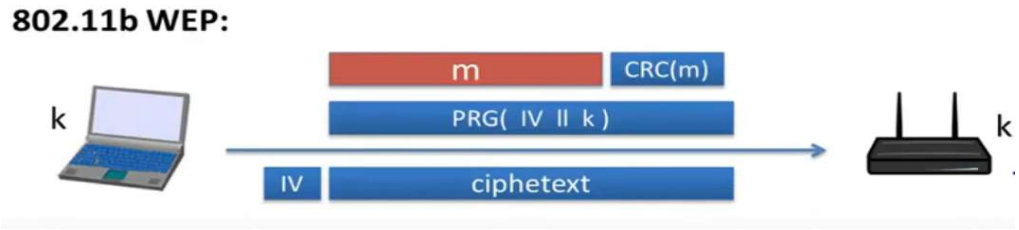


7.2. Chiffrement à flots

- Le générateur pseudo-aléatoire doit être de bonne qualité. Idéalement :
  - chaque bit de sortie vaut 0 ou 1 avec une probabilité $^1/_2$,
  - aucun bit de sortie n'est corrélé avec les précédents ou les suivants,
  - la période du générateur est suffisamment longue.
- Problème : ces trois conditions sont extrêmement difficiles à satisfaire en pratique.
- En l'absence d'une assise théorique solide, la sécurité des chiffrements à flot reste problématique. Pour preuve de cet état de fait, le nombre important d'algorithmes proposés qui ont été cassés...

7.2. Chiffrement à flots

- Le WEP (Wired Equivalent Privacy)



- Le vecteur IV est codé sur seulement 24 bits. Il est répété tous 2^{24} trames (env. 16M). Le vecteur d'initialisation est cyclique.
- Avec l'attaque des anniversaires, il suffit d'environ seulement 4822 essais pour obtenir une collision avec une probabilité de 50%.
- Sur certaines cartes 802.11, le vecteur IV est réinitialisé après un reboot.
- FMS en 2001 ont démontré qu'il fallait 1000000 de trames pour retrouver k. De nos jours, environ 40000 trames suffisent.
- Le protocole WEP n'indique pas comment est initialisé le vecteur IV. Du coup, certaines cartes WIFI l'initialisent de manière trop simple, introduisant une faiblesse supplémentaire dans le protocole.
- Aujourd'hui, pour sécuriser les sessions WIFI, on utilise le protocole WPA2 (Wi-Fi Protected Access, IEEE 802.11i).

7.2. Chiffrement à flots

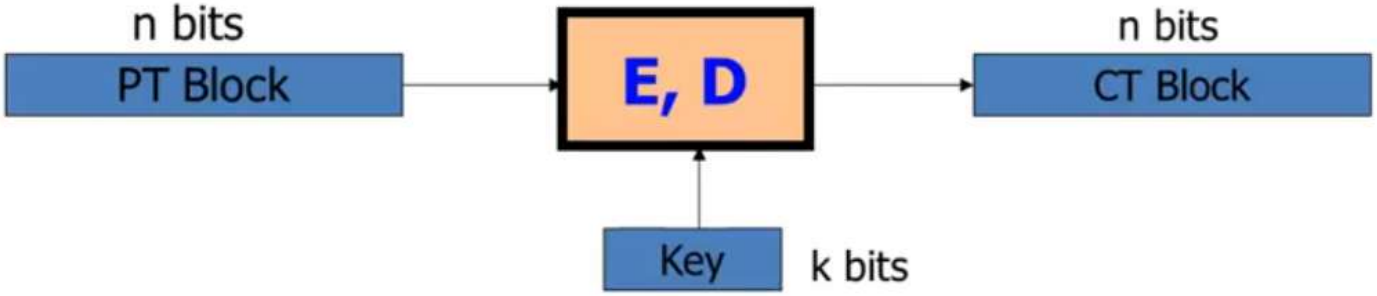
- En résumé :

Ne jamais utilisé une même clé plusieurs fois dans le chiffrement à flots.

- TLS est un protocole utilisant le chiffrement symétrique à flots. Pour qu'il soit robuste il est nécessaire que la clé soit renégociée à chaque session.
- Chiffrement de disque dur : ne jamais utiliser de chiffrement à flots.

7.3. Chiffrement par bloc

Définition

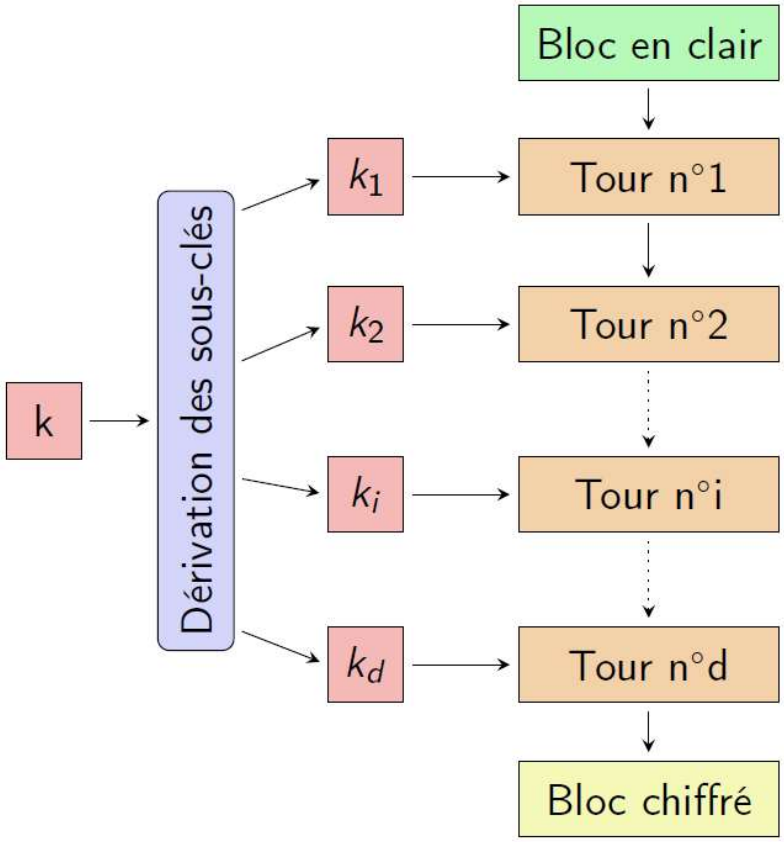


Canonical examples:

1. 3DES: n= 64 bits, k = 168 bits
2. AES: n=128 bits, k = 128, 192, 256 bits

7.3. Chiffrement par bloc

- Architecture



7.3. Chiffrement par bloc

- Chaque algorithme a son propre nombre d'itérations. Par exemple, 3DES a 48 itérations, et AES a 10 itérations.
- L'algorithme DES, standardisé en 1976, n'est plus utilisé aujourd'hui, non pas en raison d'une faiblesse structurelle (jamais découverte à ce jour), mais à cause de sa clé trop courte (56 bits, longueur imposée par la NSA, alors qu'initialement elle était de 64 bits).
- Son successeur est le triple DES ou 3DES (clé de 3 x 56 bits). Il existe plusieurs variantes dont celle recommandée par le NIST et répandue dans le monde bancaire. Elle nécessite trois (ou deux) clés.

7.4. AES (Advanced Encryption Standard)

- AES est un algorithme de chiffrement symétrique par bloc*
- AES est un sous-ensemble de la proposition initiale Rijndael : la taille des blocs est fixée à 128 bits (au lieu d'une taille variable multiple de 32 bits et comprise entre 128 bits et 256 bits).
- AES se décline en trois versions :

– **AES-128** : clé de 128 bits, 10 tours
– **AES-192** : clé de 192 bits, 12 tours
– **AES-256** : clé de 256 bits, 14 tours

7.4. AES

- Performance et sécurité
- AES-128 est presque aussi rapide que DES. Précisément, AES-128 est 2.7 fois plus rapide que 3DES, lui-même 3 fois plus lent que DES. AES est donc très rapide et en outre peu gourmand en mémoire.
- Cela lui permet d'être efficace sur une grande variété de matériels. AES a été conçu pour résister aux attaques classiques comme la cryptanalyse linéaire ou différentielle.
- Au plus la clé est longue au plus l'algorithme est sécurisé, mais au plus il est lent.

7.5. Librairie OpenSSL - PHP

```php
openssl_encrypt(
  string $data,
  string $cipher_algo,
  string $passphrase,
  int $options = 0,
  string $iv = "",
  string &$tag = null,
  string $aad = "",
  int $tag_length = 16
): string|false
```

data :Les données du message en texte brut à chiffrer.

cipher_algo : La méthode de cipher. Pour une liste des méthodes de cipher disponible, utiliser openssl_get_cipher_methods().

Passphrase : La passphrase. Si la passphrase est plus courte qu'attendu, elle est silencieusement capitonné avec des caractères NUL; si la passphrase est plus longue qu'attendu, elle est silencieusement tronqué.

Options :options est une disjonction au niveau des bits des drapeaux OPENSSL_RAW_DATA et OPENSSL_ZERO_PADDING.

iv : Un vecteur d'initialisation non-nul.

tag : Le tag d'authentification passé par référence lors de l'utilisation du mode cipher AEAD (GCM ou CCM).

aad :Données additionnelles d'authentification.

tag_length : la longueur du tag d'authentification. Sa valeur peut être entre 4 et 16 pour le mode GCM.

7.5. Librairie OpenSSL - PHP

- Exercice

1) En utilisant les algorithmes AES128, AES192, et AES256, chiffer le texte « *La cryptographie est une des disciplines de la cryptologie s'attachant à protéger des messages (assurant confidentialité, authenticité et intégrité) en s'aidant souvent de secrets ou clés. Elle se distingue de la stéganographie qui fait passer inaperçu un message dans un autre message alors que la cryptographie rend un message supposément inintelligible à autre que qui-de-droit* » (Wikipédia/Cryptographie), et afficher le message chiffré
2) Puis déchiffrer
3) Calculer le temps d'exécution de chacun des algorithmes pour 1000 itérations
4) Conclure

7.5. Librairie OpenSSL - PHP

```php
$plaintext = "La cryptographie est une des disciplines de la cryptologie s'attachant à protéger
des messages (assurant confidentialité, authenticité et intégrité) en s'aidant souvent de
secrets ou clés. Elle se distingue de la stéganographie qui fait passer inaperçu un message dans
un autre message alors que la cryptographie rend un message supposément inintelligible à autre
que qui-de-droit";
$cipher = array("AES128"=>16, "AES192"=>24, "AES256"=>32);
$j=1000;
foreach ($cipher as $algo=>$keylength){
        $total_time = 0;
        $execution_debut = microtime(true);
    for ($i=0; $i<=$j;$i++){
        $ivlen = openssl_cipher_iv_length($algo);
        $iv = openssl_random_pseudo_bytes($ivlen);
        $key = openssl_random_pseudo_bytes($keylength);
        $ciphertext = openssl_encrypt($plaintext, $algo, $key,0,$iv);
        //store $cipher, $iv, and $tag for decryption later
        $original_plaintext = openssl_decrypt($ciphertext, $algo, $key, 0,$iv);
    }
    $execution_fin = microtime(true);
    $total_time = ($execution_fin - $execution_debut);

    echo 'le ciffer '.$algo.' a exécuté '.$j.' chiffrements/déchiffrements en :'.$total_time.'
    secondes<br><br>';
}
```

La taille de la clé influe sur le temps d'exécution de l'algorithme.