

Révision secu 2022 :

Kerckhoos principle :

- 1. The system must be practically, if not mathematically, indecipherable.**
- 2. It should not require secrecy, and it should not be a problem if it falls into enemy hands.**

3. It must be possible to communicate and remember the key without using written notes, and correspondents must be able to change or modify it at will.
4. It must be applicable to telegraph communications.
5. It must be portable, and should not require several persons to handle or operate.
6. Lastly, given the circumstances in which it is to be used, the system must be easy to use and should not be stressful to use or require its users to know and comply with a long list of rules.

Classification des systèmes de cryptage :

- **Sécurité inconditionnelle** (*unconditional security* aussi appelée *perfect secrecy*):
 - La sécurité du système de cryptage n'est pas compromise par la puissance de calcul destinée à la cryptanalyse.
 - Cette catégorie s'appuie sur la théorie de l'information publiée par Shannon en 1949.
 - Plus précisément, un système de cryptage est *inconditionnellement sûr* si la probabilité de rencontrer un *plaintext* x après l'observation du *ciphertext* correspondant y est identique à la probabilité *à priori* de rencontrer le *plaintext* x . En d'autres termes, le fait de disposer de couples *plaintext/ciphertext* (x, y) ne constitue aucune aide pour la cryptanalyse.
 - Une condition nécessaire pour qu'un système soit inconditionnellement sûr est que la clé soit au moins de la même taille que le message et, surtout, qu'elle ne soit pas réutilisée pour encrypter des messages différents.
 - Cette condition rend ces systèmes peu adaptés aux besoins cryptographiques habituels et réduit leur domaine d'intérêt à un cadre théorique.
 - L'exemple classique est le *one-time pad* inventé en 1917 par J.Mauborgne and G. Vernam.
 - Fondements théoriques des systèmes inconditionnellement sûrs + d'autres exemples dans [Sti06].

- **As hard as / équivalent / provable security**

- Lorsqu'on peut prouver que la cryptanalyse de l'algorithme est aussi difficile que de résoudre un problème mathématique réputé difficile.
- Par exemple la factorisation de grands nombres, le calcul de racines carrées modulo un "composite", le calcul de logarithmes discrets dans un groupe fini, etc. (voir chapitre sur la Cryptographie Asymétrique).
- L'algorithme de Rabin et RSA (cas générique¹) sont "prouvés" équivalents à la factorisation. Une telle preuve s'appelle de "réduction" (*reduction proof*).
- La notion de *provable security* est à l'origine d'une importante controverse dans le monde cryptographique².
- **Sécurité calculatoire** (*computational security* aussi appelé *practical security*)
 - Un système de cryptage est dans cette catégorie si l'effort calculatoire nécessaire à le "casser" en utilisant les meilleures techniques possibles est au delà (avec une marge raisonnable) des ressources de calcul d'un adversaire hypothétique.
 - La grande majorité de systèmes de cryptage symétriques (AES, DES, IDEA, RC4, etc.) sont dans cette catégorie.

Entropie :

Définie par :

$$H(x) = - \sum_{i=1}^n p_i \log p_i = \sum_{i=1}^n p_i \log(1/p_i)$$

Propriétés:

- (i) $0 \leq H(X) \leq \log n$
- (ii) $H(X) = 0$ ssi $\exists i$ t.q. $p_i = 1$ et $p_j = 0, \forall j \neq i$ (c.à.d. il n'y a pas d'incertitude sur le résultat)
- (iii) $H(X) = \log n$ ssi $p_i = 1/n \forall i 1 \leq i \leq n$ (c.à.d. tous les résultats sont équiprobables).

Exemple:

L'entropie de la variable "jours de la semaine" en admettant que toutes les valeurs

sont équiprobables serait: $H(\text{jours de la semaine}) = \sum_{i=1}^7 \frac{1}{7} \log 7 = \log 7 = 2,807$

Attacks :

- Attaque ciphertext-only: L'adversaire (ou le cryptanalyste) essaye de trouver la clé ou le plaintext à partir de l'observation du ciphertext seul. Un système de cryptage vulnérable à cette attaque n'offre aucune sécurité.
- Attaque known-plaintext: L'adversaire a des couples plaintext/ciphertext à disposition. Cette attaque est à peine plus facile à mettre en place que le ciphertext-only.
- Attaque chosen-plaintext: L'adversaire peut choisir le plaintext et obtenir le ciphertext correspondant, le but étant de retrouver du plaintext à partir des ciphertexts observés.
- Attaque adaptive chosen-plaintext: Il s'agit d'une attaque chosen-plaintext où le choix sur les plaintexts peut dépendre des ciphertexts reçus lors des requêtes précédentes.
- Attaque chosen-ciphertext: L'adversaire choisit le ciphertext et obtient le plaintext correspondant. L'objectif de cette attaque étant normalement de trouver la clé.
- Attaque adaptive chosen-ciphertext: Il s'agit d'une attaque chosen-ciphertext où le choix sur les ciphertexts peut dépendre des plaintexts reçus lors des requêtes précédentes.

Random Oracle :

- A random oracle is an abstract entity (available to legitimate parties and adversaries) that responds to queries containing a given input x with perfectly random responses $Orc(x)$.
- The only exception to this behaviour resides in the previously processed inputs $(x_1, x_2, x_3, \dots, x_n)$ where the random oracle will provide the same response than the previous query so that if $x_1' = x_1$ then $Orc(x_1') = Orc(x_1)$ which means that the random oracle is deterministic with previously processed inputs.
- A random oracle can be modeled as a mathematical function $Orc: X \rightarrow Y$ where $\forall x \in X$ we have that $Pr(Orc(x) = y) = 1/\text{Sizeof}(Y) \quad \forall y \in Y$.
- A random oracle behaves like an “ideal” cryptographic hash function and as such is a valuable tool to prove security assertions under the so called *random oracle model*¹. The previous point ensures that all the possible hash function outputs (*digests*) are *equiprobable*.
- The classical case where adversaries are bounded by computational factors is called the *standard model*.
- A cryptographic protocol that is proven secure in the random oracle model may result insecure when replacing the random oracle with a “real life” hash function as SHA-1, SHA-256, etc.

- An encryption/decryption/signing oracle is also an abstract entity that will perform the corresponding operations as an “on demand” service to all legitimate and illegitimate parties.
- This entity will use (without disclosing them) the same keys as the legitimate key owners for both symmetric and asymmetric cryptosystems returning the authentic plaintext, ciphertexts or signed documents!
- Assuming E is a symmetric encryption primitive and k is the secret symmetric key, the encryption oracle will return $y = E_k(x)$ for any given input plaintext x and the decryption oracle will return x such as $E_k(x) = y$ for any given input ciphertext y .
- In Public Key Cryptosystems the oracle is only necessary for secret key operations (whether is signing or decrypting) since public key operations are openly available.
- As a result, assuming E' is a public key encryption system and pubk and privk the public and private keys, the decryption oracle will return x such as $E'_{\text{pubk}}(x) = y$ for any given input ciphertext y .
- Similarly, assuming S is a public key digital signature system and pubk and privk are respectively the validation and signing keys, the signing oracle will return y such as $S_{\text{privk}}(x) = y$ for any given input x (the data to be signed).
- *Chosen-plaintext* and (*adaptive*) *chosen-ciphertexts* attacks are normally modeled on the assumption that these oracles are available to the adversary.

Indistinguishable Ciphertexts and semantic security :

- *Ciphertext Indistinguishability* is a property of cryptographic protocols that ensures that an adversary will be unable to distinguish between different ciphertexts for given plaintexts.
- Let's assume an adversary with polynomial time computing capability and unlimited access to an encryption oracle with secret key k , proceeds to the following steps:
 - He generates two plaintexts M_0 and M_1 and sends them to the encryption oracle.
 - The encryption oracle picks an index i at random $i \in \{0,1\}$ and sends the corresponding ciphertext $c_i = E_k(M_i)$ to the adversary.
 - The adversary is free to perform any additional computations including calls to the encryption oracle for any other M_j with $j \notin \{0,1\}$
- We say that a cryptosystem is *indistinguishable under chosen plaintext attacks (IND-CPA)* if such an adversary has a negligible advantage over random guessing to successfully picking the right index i ($\text{Prob} = 1/2 + \epsilon$ with ϵ small and bound).
- It should be noted that for public key cryptosystems the presence of an encryption oracle is unnecessary since encryptions involves public keys and consequently can be easily performed by the adversary.
- Cryptosystems featuring IND-CPA are also said to provide *semantic security*¹.

Deterministic Encryption :

- Cryptographic algorithms display a deterministic behaviour since encryption and decryption operations yield the same results over identical inputs.
- This may result in clear weaknesses in terms of ciphertext indistinguishability if an encryption oracle is available or if public key cryptosystems are used.
- As an example, let's assume Alice sends a simple message (i.e. 'yes' or 'no') to Bob encrypted with Bob's public key. If the adversary can guess the semantics of the message, he could easily compute the corresponding ciphertexts $C_{\text{yes}} = E_{\text{Bobpubkey}}(\text{'Yes'})$ and $C_{\text{non}} = E_{\text{Bobpubkey}}(\text{'Non'})$.
- If no extra random is added before encryption both ciphertexts are clearly distinguishable with $\text{Prob} = 1$ and, as a result, plaintexts would be disclosed without knowledge of Bob's private key.
- The adversary can even build a codebook of known plaintexts made of semantically sound messages and compare them with observed ciphertexts for eventual matches without cryptanalysing the algorithm!
- This problem also applies to symmetric cryptosystems provided that an encryption oracle is available and that no random Initialization Vector (IV) is used (more on this subject in the section describing block ciphers).

Probabilistic Encryption :

- Consists in adding randomness to the cryptosystem by processing the plaintext before applying the encryption function. As a result, when dealing with multiple encryption instances the same plaintext will result in different ciphertexts.
- Building semantically sound codebooks becomes useless for adversaries. The final aim being to obtain ciphertext indistinguishability and semantic security for public-key cryptosystems.
- Initial probabilistic encryption approaches¹ had unpractical message expansion factors.
- The most commonly used solution is *Optimal Asymmetric Encryption Padding*² (OAEP) where the initial plaintext P is combined with a hashed random number R as follows:
$$M_1 := P \oplus h(R) \text{ and } M_2 := R \oplus h(M_1).$$

M₁ and M₂ are then encrypted: C₁ = E_{pubk}(M₁) and C₂ = E_{pubk}(M₂) and sent.
Upon decryption R is computed: R = M₂ ⊕ h(M₁) and then P: P = h(R) ⊕ M₁.
- The security proofs provided in the initial OAEP publication have been questioned in recent papers³. OAEP is the basis of RSA-PKCS1 encryption standard.

Le One-Time Pad :

Soit $n \geq 1$ et les espaces P, C, K resp. des plaintexts, ciphertexts et clés possibles tels que: $P, C, K = (Z_2)^n$. Soient $x = (x_1, x_2, \dots, x_n) \in X$, $y = (y_1, y_2, \dots, y_n) \in Y$ et $k = (k_1, k_2, \dots, k_n) \in K$. Alors, les opérations d'encryption et decryption d'un *one-time pad* (aussi appelé *Vernam Cipher*) sont définies comme suit:

$$\begin{aligned} E_k(x_i) &= x_i \oplus k_i & 1 \leq i \leq n \\ D_k(y_i) &= y_i \oplus k_i & 1 \leq i \leq n \end{aligned}$$

- Si les k_i sont choisis de manière indépendante et aléatoire, le *one-time pad* est unconditionnellement sûr contre des attaques *ciphertext-only* ce qui veut dire que le fait d'observer des ciphertexts n'est d'aucune aide pour la cryptanalyse, ou dans d'autres termes, que l'entropie de X n'est pas diminuée après l'observation des ciphertexts, soit: $H(X|C) = H(X)$. ($H(X)$ = fonction d'entropie, $H(X|C)$ = entropie conditionnelle)
Ceci reste vrai même si l'adversaire a des ressources de calcul infinies!
- Problème: Shannon a prouvé qu'une condition nécessaire pour qu'un système de cryptage à clé symétrique soit inconditionnellement sûr est que $H(K) \geq H(X)$. En admettant que les composants d'une clé de m bits sont aléatoires et équiprobables, on a que $H(K) = m$ et donc que $m \geq H(X)$. Donc pour satisfaire l'hypothèse de Shannon indépendamment de l'entropie de X , il faut que la longueur de la clé aléatoire soit au moins aussi grande que celle du plaintext!
- Une première conséquence de cette propriété est que la clé ne peut (même partiellement) être réutilisée. Voyons le résultat de la réutilisation de la même clé sur deux plaintexts différents:

$$\begin{aligned} E_k(x_a) &= y_a = x_a \oplus k \\ E_k(x_b) &= y_b = x_b \oplus k \\ y_a \oplus y_b &= x_a \oplus k \oplus x_b \oplus k = \boxed{x_a \oplus x_b !!!} \end{aligned}$$

ce qui avec des plaintexts de faible entropie permet de retrouver les deux plaintexts et même de calculer la clé k car:

$$\boxed{k = y_a \oplus x_a}$$

- Ceci signifie que le *one-time pad* est vulnérable à l'attaque *known plaintext*, même si ceci n'a pas d'importance si une nouvelle clé est regénérée pour chaque nouveau message.
- La contrainte sur la longueur de la clé pose un problème évident dans la distribution et la gestion des clés. De ce fait l'utilisation réelle du *one-time pad* est très peu significative.
- L'avènement de la *cryptographie quantique* proposant des canaux confidentiels de distribution de clés de longueur illimitée a relancé l'intérêt du *one-time pad* mais l'application de ces techniques aux réseaux classiques des télécommunications est pour le moment impossible.

Stream cipher :

- Les *stream ciphers* constituent une famille de systèmes de cryptage où la taille du bloc encrypté est égale à 1 bit.
- Les *stream ciphers* sont généralement composés de deux phases:
 - Une phase de génération de la séquence d'éléments formant la clé (le *keystream*). ①
 - Une phase de substitution où les bits du plaintext subissent une opération spécifique dépendante du *keystream*. ②
- Un exemple évident d'un *stream cipher* est le *one-time pad* avec:
 - Une phase de génération du *keystream* effectuée par un générateur (*pséudo-*) aléatoire.
 - Une phase de substitution qui consiste à effectuer un **xor** (\oplus) avec le *keystream*.

Stream Ciphers: Caractéristiques

- Rapidité: Le cryptage se fait directement au niveau des registres. Idéal pour des applications nécessitant un cryptage "on the fly" comme le *video streaming*.
- Facilité: Les opérations peuvent être effectuées par des systèmes ayant des ressources CPU limitées.
- Pas (ou peu...) besoin de mémoire/buffering.
- Propagation des erreurs limitée ou absente: la retransmission des paquets fautifs suffit normalement (adapté aux applications où les pertes de paquets sont fréquentes comme les transmissions sans fil (*WiFi*)).
- Inconvénients:
 - La qualité en termes de randomness du keystream générée détermine la robustesse du système.
 - La réutilisation du keystream permet une cryptanalyse facile (cf. le *one-time pad*).

SC synchrone :

- Le keystream généré dépend seulement de la clé et non pas du plaintext ni du ciphertext.
- Le processus d'encryption d'un stream cipher synchrone est décrit par les équations suivantes:

$$\sigma_{i+1} = f(\sigma_i, k) \quad z_i = g(\sigma_i, k) \quad c_i = h(z_i, m_i)$$

avec σ_i l'état initial qui peut dépendre de la clé k , f la fonction qui détermine l'état suivant, g la fonction qui produit le keystream z_i et h la fonction de sortie qui produit le ciphertext c_i à partir du plaintext m_i .

- Schématiquement, le mode de fonctionnement d'un stream cipher synchrone est le suivant:

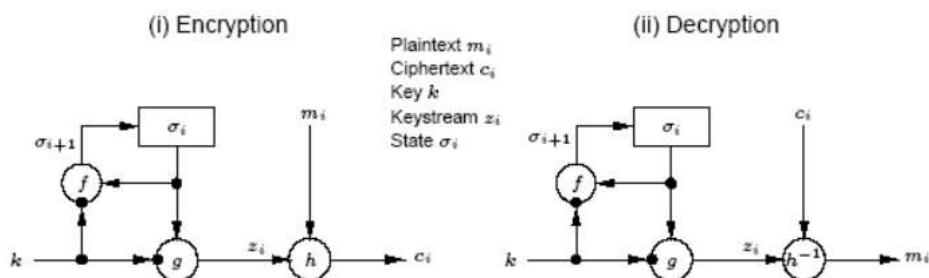
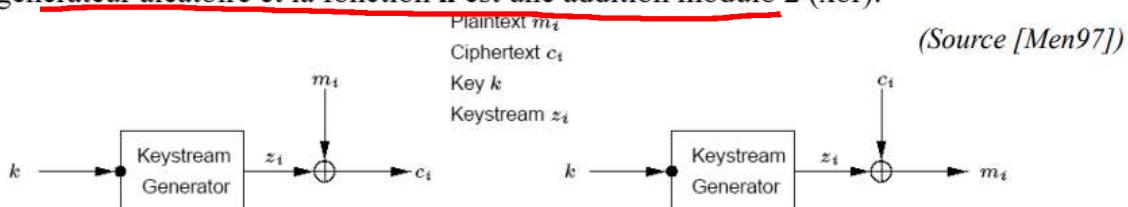


Figure 6.1: General model of a synchronous stream cipher.

- Nécessitent la synchronisation de l'émetteur et du récepteur: En plus d'utiliser la même clé k , les deux doivent se trouver dans le même état pour que le processus fonctionne. Si la synchronisation est perdue il faut des mécanismes externes pour la récupérer (marqueurs spéciaux, analyses de redondance du plaintext, etc.)
- Pas de propagation d'erreur. La modification du ciphertext pendant la transmission n'entraîne pas des perturbations dans des séquences de ciphertext ultérieures (cependant, la suppression d'un ciphertext provoquerait la désynchronisation du récepteur).
- Attaques actives: l'insertion, l'élimination ou le *replay* de parties de ciphertext sont détectés par le récepteur. Cependant, un adversaire pourrait modifier certains bits du ciphertext et analyser l'impact sur le plaintext correspondant. Des mécanismes d'authentification d'origine supplémentaires sont nécessaires afin de détecter ces attaques.
- Cas les plus fréquent des Stream Cipher Syncrone: le stream cipher additif (cf. *le one-time pad*) où les fonctions f et g générant le keystream sont remplacées par un générateur aléatoire et la fonction h est une addition modulo 2 (xor):



SC Asynchrone :

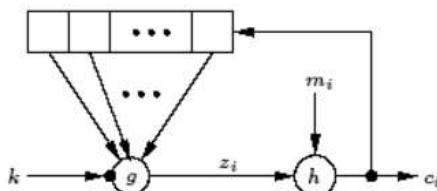
- Aussi appelés **auto-synchronisés** (*self synchronizing ciphers*).
- Le keystream généré dépend de la clé ainsi que d'un nombre fixé de ciphertexts précédents.
- Le processus d'encryption d'un *stream cipher asynchrone* est décrit par les équations suivantes:

$$\sigma_i = (c_{i-t}, c_{i-t+1}, \dots, c_{i-1}) \quad z_i = g(\sigma_i, k) \quad c_i = h(z_i, m_i)$$

avec σ_i , g et h comme pour le cas synchrone.

- Schématiquement, le mode de fonctionnement d'un *stream cipher asynchrone* est le suivant:

(i) Encryption



(ii) Decryption

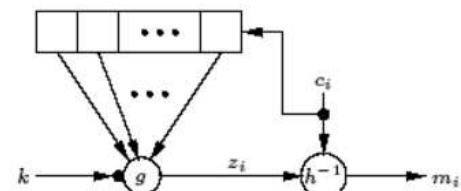


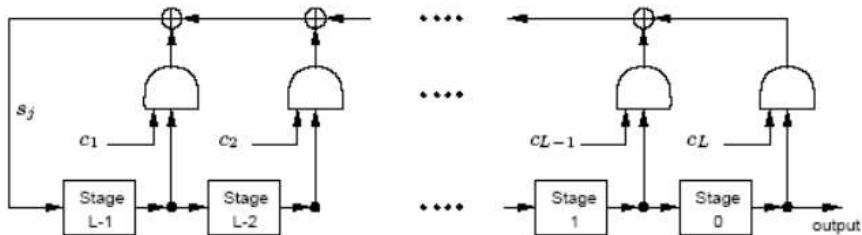
Figure 6.3: General model of a self-synchronizing stream cipher.

- **Auto-synchronisation:** En cas d'élimination ou d'insertion de ciphertexts en cours de route, le récepteur est capable de se *re-synchroniser* avec l'émetteur grâce à la mémorisation (buffer) d'un nombre de ciphertext précédents.
- **Propagation d'erreurs limitée:** La propagation d'erreurs s'étend uniquement au nombre de bits du ciphertext mémorisés (taille du *buffer*). Après, la decryption se déroule à nouveau correctement.
- **Attaques actives:** La modification de fragments du ciphertext sera plus facilement détecté que dans le cas synchrone à cause de la propagation d'erreurs. Cependant, comme le récepteur est capable de s'auto-synchroniser avec l'émetteur, même si des ciphertexts sont éliminés ou insérés en cours de route, il convient de vérifier l'intégrité et l'authenticité du flot entier.
- **Diffusion des statistiques du plaintext:** Le fait que chaque bit du plaintext aura une influence sur la totalité des ciphertexts subséquents se traduit par une plus grande dispersion des statistiques du plaintext comparée au cas synchrone...
- ... Il convient, donc, d'utiliser des stream ciphers asynchrones lorsque l'entropie des plaintexts est limitée et pourrait permettre des attaques ciblées aux plaintexts fortement redondants.

Générateur de keystream :

- Lorsqu'il convient de générer un keystream d'une longueur \mathbf{m} à partir d'une clé secrète de longueur \mathbf{l} avec $\mathbf{l} \ll \mathbf{m}$, on fait appel à des générateurs de keystreams.
- Le plus courant de ces générateurs est le *Linear Feedback Shift Register* (LSFR).
- Un LSFR a les caractéristiques suivantes:
 - S'adapte très bien aux implantations hardware.
 - Produit des séquences de périodes longues et avec une qualité aléatoire notable (*randomness* assez forte)
 - Se base sur les propriétés algébriques des combinaisons linéaires.
- Exemple générique d'un LSFR de longueur \mathbf{L} :

Figure 6.4 depicts an LFSR. Referring to the figure, each c_i is either 0 or 1; the closed semi-circles are AND gates; and the feedback bit s_j is the modulo 2 sum of the contents of those stages i , $0 \leq i \leq L - 1$, for which $c_{L-i} = 1$.

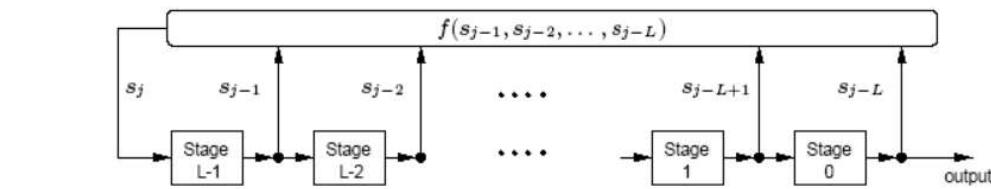


(Source [Men97])

Figure 6.4: A linear feedback shift register (LFSR) of length L .

LSFRs: Quelques Remarques

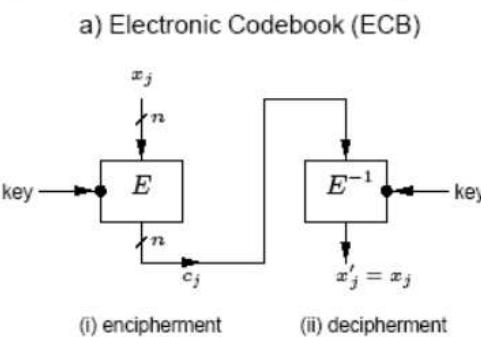
- Les LSFRs sont des constructions très répandues dans la cryptographie et dans la théorie de codes.
- Un grand nombre de *stream ciphers basés sur les LSFRs* (surtout dans la sphère militaire) ont été développés dans le passé.
- Malheureusement, le niveau de sécurité offert par ces systèmes est jugé insuffisant de nos jours (comparé à celui des *blocks ciphers*...)
- La métrique permettant d'analyser un LFSR est sa *complexité linéaire* (**linear complexity**). L'algorithme de *Berlekamp-Massey*¹ permet de déterminer la complexité linéaire d'un LSFR et de calculer ainsi un nombre arbitrairement grand de séquences générées par un LSFR.
- Une solution pour augmenter la complexité est de substituer la combinaison linéaire des bits du ciphertext par une fonction non linéaire f . Ce sont les *Non Linear Feedback Shift Registers*:



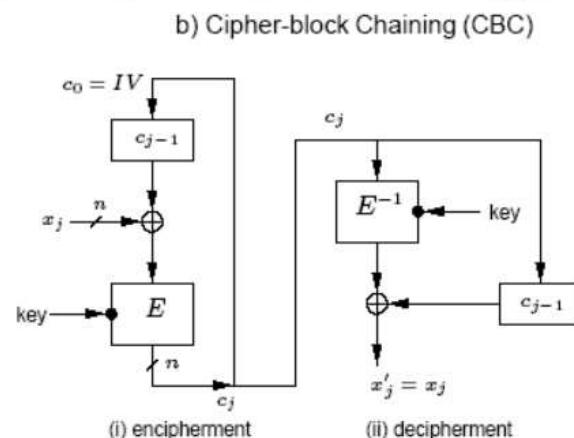
Block Cipher :

- Les *block ciphers* symétriques constituent la pierre angulaire de la cryptographie. Leur fonctionnalité principale est la confidentialité mais ils sont également à la base des services d'*authentification, fonctions de hachage, génération aléatoire*, etc.
- Définition: Un **block cipher** est une **fonction** qui fait correspondre à un bloc de n-bits un autre bloc de la même taille. La fonction est paramétrée par une clé K de k-bits. Afin de permettre une decryption unique, la fonction doit être **bijective**. *Chaque clé définit une bijection différente*. La taille d'entrée du bloc sur lequel s'applique l'encryption s'appelle aussi **taille nominale** de l'algorithme.
- Critères pour évaluer la qualité d'un *block cipher*:
 - **Taille/Entropie de la clé:** Idéalement, les clés sont équiprobables et l'espace des clés a une entropie égale à k. Une forte entropie de la clé protège des attaques *brute-force* à partir de *chosen/known plaintexts*. Les *block ciphers* modernes doivent avoir des clés d'au moins 128 bits.
 - **Performances**
 - **Taille du bloc:** Un bloc trop petit permettrait des attaques où des “dictionnaires” plaintext/ciphertext seraient construits. De nos jours, des blocs de taille ≥ 128 bits deviennent courants.
 - **Résistance cryptographique:** Le *block cipher* doit se montrer résistant à des techniques de cryptanalyse connues: *cryptanalyse linéaire ou différentielle, meet in the middle*, etc. L'effort inhérent à ces attaques (complexité, stockage, parallélisation, etc.) doit être équivalent à celui d'une attaque *brute force*.

Electronic Codebook (ECB)



Cipher-block Chaining(CBC)

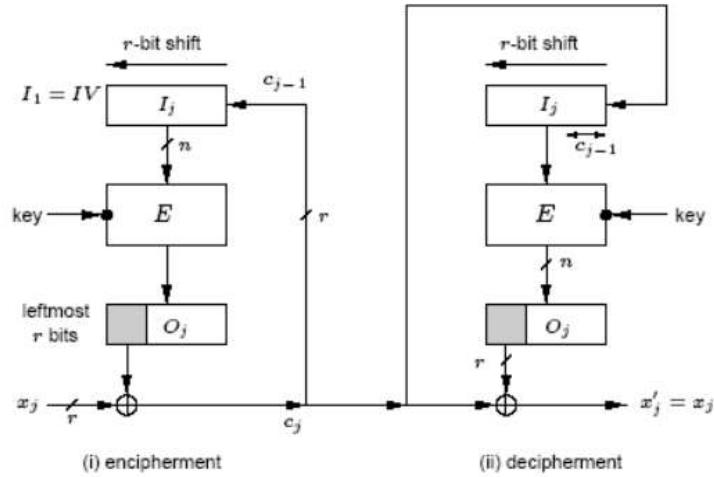


- Des plaintexts identiques donnent lieu à des ciphertexts identiques
- Pas de propagation d'erreurs sur les blocs adjacents

- Des plaintext identiques donnent lieu à des ciphertexts différents si le **IV** change
- Les *patterns* du plaintext sont “effacées” du ciphertext (chaînage)
- Une erreur sur c_j affecte uniquement la décription des blocs c_j et c_{j+1}

Cipher Feedback Mode: CFB (Source [Men97])

c) Cipher feedback (CFB), r -bit characters/ r -bit feedback



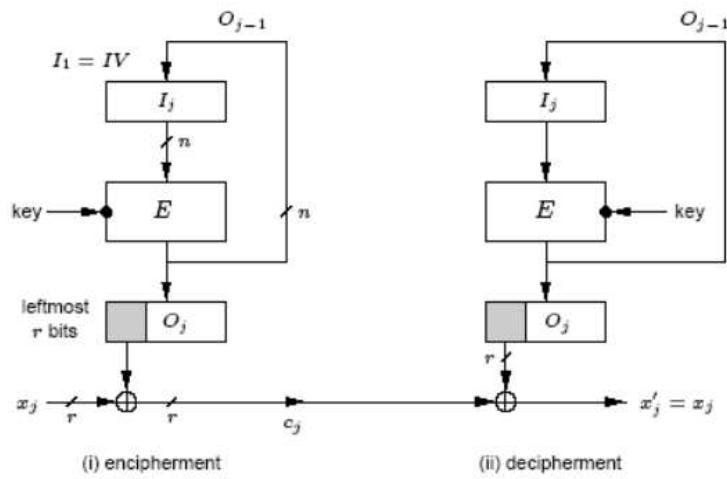
7.17 Algorithm CFB mode of operation (CFB-r)

INPUT: k -bit key K ; n -bit IV ; r -bit plaintext blocks x_1, \dots, x_u ($1 \leq r \leq n$).
SUMMARY: produce r -bit ciphertext blocks c_1, \dots, c_u ; decrypt to recover plaintext.

1. Encryption: $I_1 \leftarrow IV$. (I_j is the input value in a shift register.) For $1 \leq j \leq u$:
 - (a) $O_j \leftarrow E_K(I_j)$. (Compute the block cipher output.)
 - (b) $t_j \leftarrow$ the r leftmost bits of O_j . (Assume the leftmost is identified as bit 1.)
 - (c) $c_j \leftarrow x_j \oplus t_j$. (Transmit the r -bit ciphertext block c_j .)
 - (d) $I_{j+1} \leftarrow 2^r \cdot I_j + c_j \bmod 2^n$. (Shift c_j into right end of shift register.)
2. Decryption: $I_1 \leftarrow IV$. For $1 \leq j \leq u$, upon receiving c_j :
 $x_j \leftarrow c_j \oplus t_j$, where t_j , O_j and I_j are computed as above.

Output Feedback Mode: OFB (Source [Men97])

d) Output feedback (OFB), r -bit characters/ n -bit feedback



7.20 Algorithm OFB mode with full feedback (per ISO 10116)

INPUT: k -bit key K ; n -bit IV ; r -bit plaintext blocks x_1, \dots, x_u ($1 \leq r \leq n$).
SUMMARY: produce r -bit ciphertext blocks c_1, \dots, c_u ; decrypt to recover plaintext.

1. Encryption: $I_1 \leftarrow IV$. For $1 \leq j \leq u$, given plaintext block x_j :
 - (a) $O_j \leftarrow E_K(I_j)$. (Compute the block cipher output.)
 - (b) $t_j \leftarrow$ the r leftmost bits of O_j . (Assume the leftmost is identified as bit 1.)
 - (c) $c_j \leftarrow x_j \oplus t_j$. (Transmit the r -bit ciphertext block c_j .)
 - (d) $I_{j+1} \leftarrow O_j$. (Update the block cipher input for the next block.)
2. Decryption: $I_1 \leftarrow IV$. For $1 \leq j \leq u$, upon receiving c_j :
 $x_j \leftarrow c_j \oplus t_j$, where t_j , O_j , and I_j are computed as above.

Modes CFB et OFB: Caractéristiques

Les modes CFB et OFB fonctionnent comme un *stream cipher* avec un *keystream* généré par le bloc de cryptage. Dans CFB, le *keystream* dépend des ciphertexts précédents (asynchrone) alors que dans OFB, le *keystream* est entièrement déterminé par la clé et le *IV* (synchrone).

Particularités de CFB:

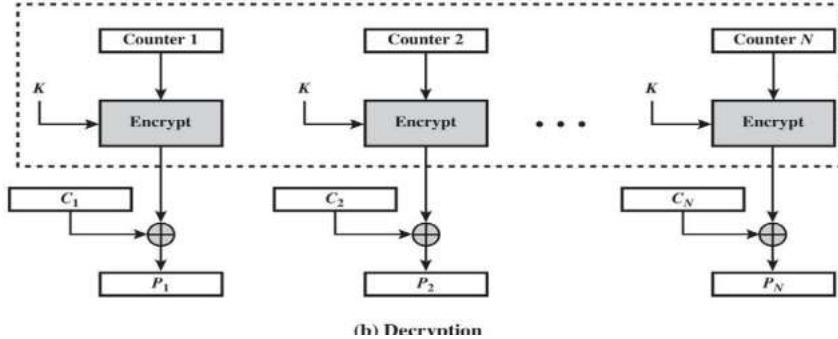
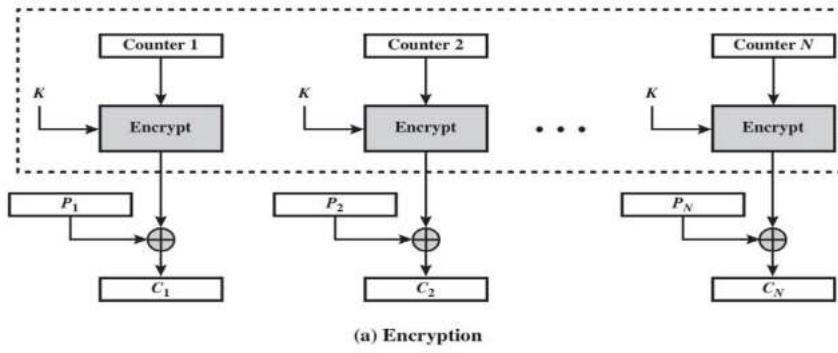
- Comme dans le mode CBC, des plaintext identiques sont traduits en ciphertexts différents si le IV change. Le IV n'est pas nécessairement confidentiel et peut être échangé en clair entre les parties.
- Le chaînage introduit également des dépendances entre les ciphertexts courants et les ciphertexts précédents. En particulier, si n est la taille nominal de l'algorithme et r est la taille des plaintexts, le ciphertext courant dépendra des n/r ciphertexts précédents (chaque itération décalera l'entrée fautive de r positions, après n/r itérations le ciphertext fautif sera "expulsé" complètement).
- La propagation d'erreurs obéit au même principe: une erreur dans un ciphertext se traduira par une mauvaise decryption des n/r ciphertexts suivants.

Particularités de OFB:

- OFB a un comportement identique aux modes CBC et CFB pour l'encryption de plaintext identiques.
- Pas de propagation d'erreurs sur les ciphertexts adjacents.
- Modifiez le IV si la clé ne change pas pour éviter la réutilisation du *keystream*!!!

Counter Mode (CTR Mode)

Fréquemment utilisé comme support d'encryption dans des protocoles de transfert de données comme ATM (*Asynchronous Transfer Mode*) et IPsec (*IP security*)



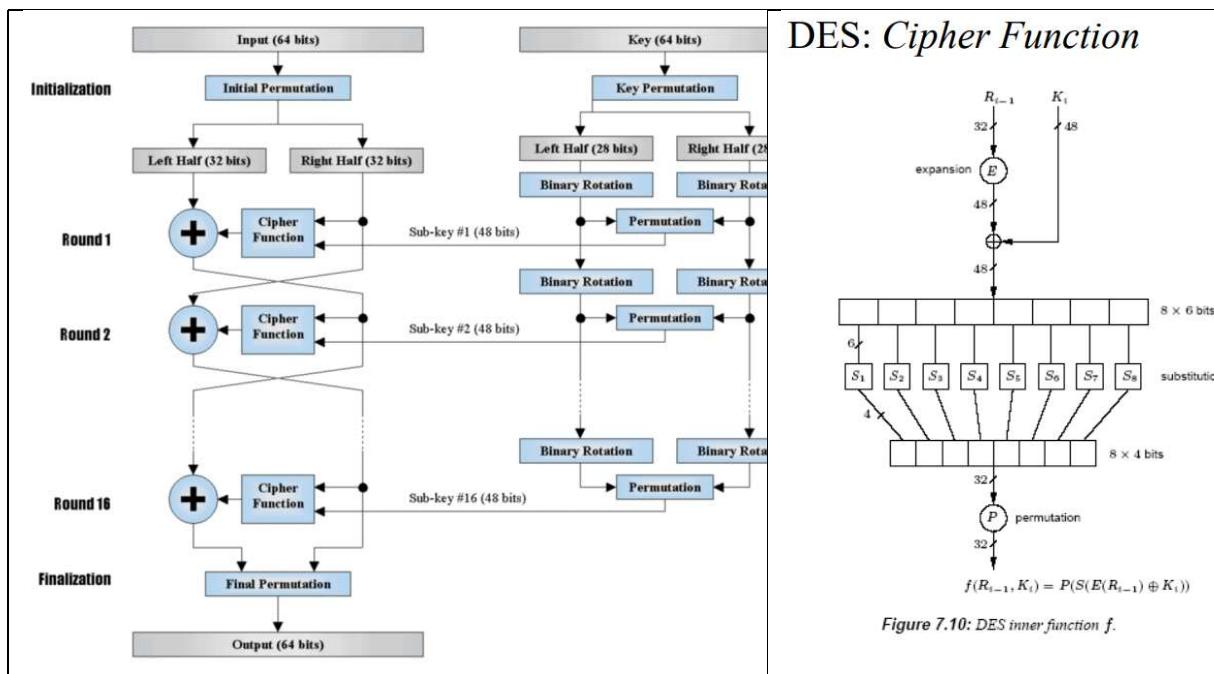
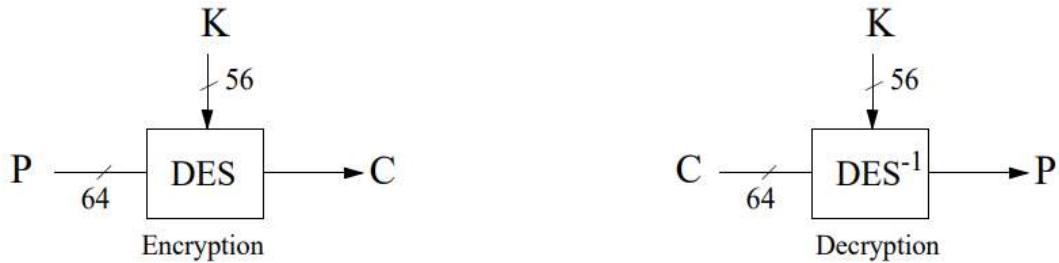
- Le *keystream* est généré par l'encryption d'un compteur aléatoire de taille 2^b (avec b la taille du bloc) et nécessaire pour la décription. Ce compteur est incrémenté modulo 2^b après chaque itération.
- Travaille en mode synchrone. La réutilisation d'un même compteur se traduit par un *keystream identique!*
- Solution: Toujours incrémenter le compteur pour chaque flot encrypté de telle sorte que le compteur du premier bloc d'un flot soit plus grand que le dernier bloc du flot précédent.
- *Facilement parallélisable:* Le keystream peut être pré-calculé aussi bien pour l'encryption que pour la décription. Profite pleinement des architectures SIMD car contrairement aux autres modes de chaînage il n'y a pas des dépendances entre les opérations des différents blocs.
- *Accès aléatoire à l'encryption/décription de chaque bloc:* Contrairement aux autres modes de chaînage où la i -ème opération dépend de la $(i-1)$ -ème opération.
- Si à ceci on ajoute l'*absence de propagation d'erreurs*, le mode compteur facilite la (re)transmission sélective des blocs de ciphertext, ce qui le rend très attractif pour la sécurisation de lignes à haut débit ainsi que pour les transferts encryptés de grands volumes d'information (p.ex. vidéo).

Product Ciphers et Feistel Ciphers

- Un ***product cipher*** est un schéma de cryptage combinant une série de transformations successives dans le but de renforcer la résistance à la cryptanalyse. Des transformations courantes pour un *product cipher* sont: des transpositions, des substitutions, des XORs, des combinaisons linéaires, des multiplications modulaires, etc.
- Un ***Feistel cipher*** est un *product cipher* itératif capable de transformer un plaintext de $2t$ bits de la forme (L_0, R_0) composé par deux sous-blocs L_0 et R_0 de t bits en un ciphertext de taille $2t$ de la forme (R_r, L_r) après r étapes (*rounds*) successives avec $r \geq 1$. Chaque étape définit une *bijection* (inversible!) pour permettre une decryption unique.
- Des *permutations* et des *substitutions* sont les opérations les plus fréquentes.
- Les étapes $1 \leq i \leq r$ s'écrivent: $(L_{i-1}, R_{i-1}) \xrightarrow{K_i} (L_i, R_i)$ avec $L_i = R_{i-1}$ et $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$. Les K_i sont des sous-clés, différentes pour chaque étape, générées à partir de la clé principale K du schéma de cryptage.
- Le nombre d'étapes propres à un *Feistel cipher* est normalement pair et ≥ 3 (p.ex. DES a 16 étapes)
- Après l'exécution de toutes les étapes, un *Feistel cipher* effectue une permutation des deux parties (L_r, R_r) en (R_r, L_r) .
- La decryption d'un *Feistel Cipher* est identique à l'encryption sauf que les sous-clés K_i sont appliquées en ordre inverse (De K_r à K_1).

Data Encryption Standard (DES)

- DES a été l'algorithme cryptographique le plus important jusqu'à l'avènement d'AES en 2001.
- DES est un *Feistel Cipher* avec des blocs de 64 bits (taille nominale).
- La taille effective de la clé est de 56 bits (Un total de 64 bits avec 8 bits de parité).
- L'algorithme est constitué de **16** étapes avec **16** sous-clés de 48 bits générées (une clé par étape).
- DES (comme tout autre *block cipher*) peut être utilisé dans les 4 modes: ECB, CBC, CFB et OFB.
- Le schéma de fonctionnement de DES est le suivant:



Fonctionement :

Cipher Fonction

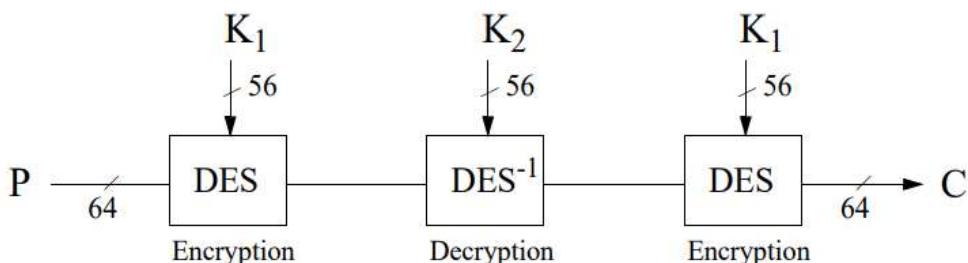
- **Expansion E:** Les 32 bits de l'entrée sont transformés en un vecteur de 48 bits en utilisant la table **E** (page 77). La première ligne de cette table indique comment sera généré le premier sous-bloc de 6 bits: on prendra en premier le 32^e bit et après les bits 1,2,3,4,5. Le deuxième sous-bloc commence par le 4^e bit ensuite les bits 5,6,7,8,9 et ainsi de suite...
- **Key addition:** XOR du vecteur de 48 bits avec la clé.
- **S-boxes:** On applique **8 S-boxes** sur le vecteur de 48 bits résultant du XOR précédent. Chacune de ces S-boxes prend un sous-bloc de 6 bits et le transforme en un sous-bloc de 4 bits. Les S-boxes 1 et 2 sont présentées en page 78. L'opération s'effectue de la manière suivante: Si on dénote les 6 bits d'input de la S-box comme: $a_1a_2a_3a_4a_5a_6$. La sortie est donnée par le contenu de la cellule située dans la ligne $a_1 + 2a_6$ et la colonne $a_2 + 2a_3 + 4a_4 + 8a_5$.
- **Permutation P:** La permutation P (page 77) fonctionne comme suit: Le premier bit est envoyé à la 16^e position, le deuxième à la 7^e position et ainsi de suite.

Permutations IP et IP^{-1}

- Agissent respectivement au début et à la fin du traitement du bloc et sur l'ensemble des 64 bits (voir les tables en page 77 pour les détails).

DES et Triple-DES

- La taille de l'ensemble de clés ($\{0,1\}^{56}$) constitue la plus grande menace qui pèse sur DES avec les ressources de calcul actuels. En 1999 il a suffit de 24 heures pour trouver la clé à partir d'un *known plaintext* en utilisant une technique brute force massivement parallèle (100'000 PCs connectés sur Internet)¹.
- **Triple DES** nous met à l'abri de ces attaques *brute force* en augmentant l'espace des clés possibles à $\{0,1\}^{112}$. Schématiquement, il fonctionne de la manière suivante:



- Cette alternative permet de continuer à utiliser les “boîtes” DES (hardware et software) en attendant une migration vers AES.
- Le niveau de sécurité obtenu par cette solution est très satisfaisant.
- L'impact en termes de performances de trois exécutions successives de DES reste un inconvénient pour certaines applications.

Des Propriétés :

- **DES n'est pas un groupe** (au sens algébrique) avec la composition: En d'autres termes, DES étant une permutation: $\{0,1\}^{64} \rightarrow \{0,1\}^{64}$, si DES était un groupe pour la composition, ceci voudrait dire que:

$$\forall (K_1, K_2), \exists K_3 \text{ t.q. } E_{K_2}(E_{K_1}(x)) = E_{K_3}(x)$$

Cette propriété permet d'assurer que l'encryption composée (comme *Triple-DES*) augmente considérablement la sécurité de DES. Si DES était un groupe, la recherche exhaustive sur l'ensemble de clés possibles ($\{0,1\}^{56}$) permettrait de "casser" l'algorithme indépendamment du nombre d'exécutions consécutives de DES.

- **Clés faibles et mi-faibles (weak and semi-weak keys):**

- Une clé **K** est dite **faible** si $E_K(E_K(x)) = x$.
- Une paire de clés (K_1, K_2) est dite **mi-faible** si $E_{K_1}(E_{K_2}(x)) = x$.
- Les clés faibles ont la particularité de générer de sous-clés identiques par paires ($k_1 = k_{16}$, $k_2 = k_{15}$, $k_8 = k_9$), ce qui facilite la cryptanalyse.
- DES a 4 clés faibles (et 6 paires de clés mi-faibles):

0101 0101 0101 0101

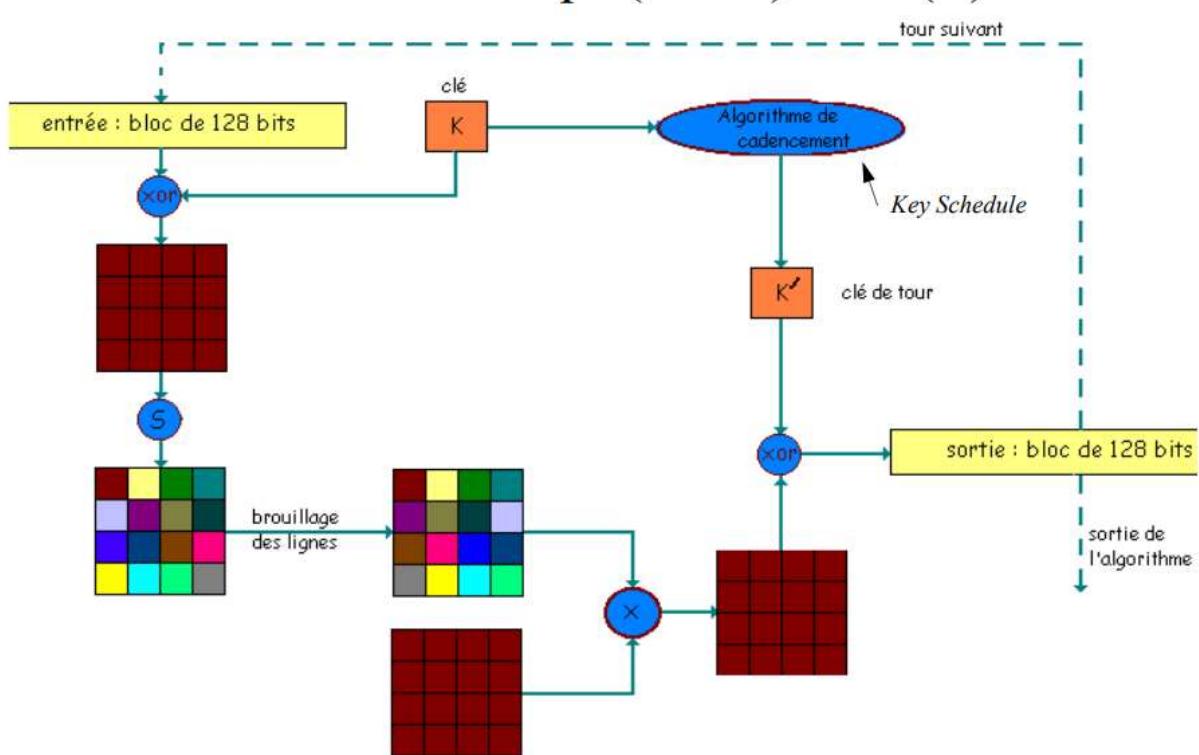
Clés faibles pour DES: 0101 0101 FEFE FEFE
FEFE FEFE FEFE FEFE
FEFE FEFE 0101 0101

AES :

- Adopté comme standard en Novembre 2001¹, conçu par *Johan Daemen et Vincent Rijmen*² (d'où son nom original *Rijndael*).
- Il s'agit également d'un *block cipher itératif* (comme DES) mais pas d'un *Feistel Cipher*
 - Blocs *Plaintext/Ciphertext*: 128 bits.
 - Clé de longueur variable: 128, 192, ou 256 bits.
- Contrairement à DES, AES est issu d'un processus de consultation et d'analyse ouvert à des experts mondiaux.
- Techniques semblables à DES (substitutions, permutations, XOR...) complémentées par des opérations algébriques simples et très performantes.
- Toutes les opérations s'effectuent dans le corps GF(2⁸): le corps fini de polynômes de degré ≤ 7 avec des coefficients dans GF(2).
- En particulier, un byte pour AES est un élément dans GF(2⁸) et les opérations sur les bytes (additions, multiplications,...) sont définies comme sur GF(2⁸).
- ~2 fois plus performant (en software) et ~10²² fois (en théorie...) plus sûr que DES...
- Évolutif: La taille de la clé peut être augmentée si nécessaire.

L'unité de base sur laquelle s'appliquent les calculs est une matrice de 4 lignes et 4 colonnes (dans le cas d'une clé de 128 bits) dont les éléments sont des bytes:

- **ByteSub:** Opération non linéaire (S-box)
conçu pour résister à la cryptanalyse
linéaire et différentielle.
- **ShiftRow:** Permutation des bytes
introduisant des décalages variables sur
les lignes.
- **MixColumn:** Chaque colonne est
remplacée par des combinaisons linéaires
des autres colonnes (multiplication des
matrices!)
- **AddRoundKey:** XOR de la matrice
courante avec la sous-clé correspondante
à l'étape courante.



- Le nombre d'étapes d'AES varie en fonction de la taille de la clé. Pour une clé de 128 bits, il faut effectuer 10 étapes. Chaque augmentation de 32 bits sur la taille de la clé, entraîne une étape supplémentaire (14 étapes pour des clés de 256 bits).
 - La decryption consiste en appliquer les opérations inverses dans chacune des étapes (*InvSubBytes*, *InvShiftRows*, *InvMixColumns*). *AddRoundKey* (à cause du XOR) est sa propre inverse.
 - Le **Key Schedule** consiste en:
 - Une opération d'expansion de la clé principale. Si N_e est le nombre d'étapes (dépendant de la clé), une matrice de 4 lignes et $4 * (N_e + 1)$ colonnes est générée.
 - Une opération de sélection de la clé d'étape: La première sous-clé sera constituée des 4 premières colonnes de la matrice générée lors de l'expansion et ainsi de suite.
 - *Pseudo-code pour AES:*
- ```

Rijndael(State,CipherKey)
{
 KeyExpansion(CipherKey,ExpandedKey); // Key Schedule
 AddRoundKey(State,ExpandedKey[0..3]); // Premier XOR avec la 1ère sous-clé
 For(i=1 ; i<Ne ; i++) Round(State,ExpandedKey[4*i ... (4*i)+3]); // Ne - 1 étapes
 FinalRound(State,ExpandedKey[4*Ne ... 4*Ne+3]). // Dernière étape (pas de MixColumn)
}

```

---

## AES: Remarques Finales et Attaques (I)

- La plus grande force de AES réside dans sa simplicité et dans ses performances, y compris sur des plate-formes à capacité de calcul réduite (p.ex. des cartes à puces avec des processeurs à 8 bits).
  - Depuis sa publication officielle, des nombreux travaux de cryptanalyse ont été publiés avec des résultats très intéressants. En particulier, N. Courtois et P.Pieprzyk<sup>1</sup> ont présenté une technique appelée XSL permettant de représenter AES comme un système de 8000 équations quadratiques avec 1600 inconnues binaires. L'effort nécessaire pour casser ce système est estimé (il s'agit encore d'une conjecture...) à  $2^{100}$ .
  - Ces attaques se basent sur le caractère fortement algébrique (et largement contesté...) de AES. De plus, il suffit de quelques *known plaintexts* pour les mettre en place, ce qui les distingue des attaques linéaires et différentielles (page 88).
  - Ces dernières années (2009-2011) des attaques basées sur des clés similaires (*related key attacks*)<sup>2</sup> ont obtenu des résultats intéressants sur des versions réduites d'AES.
  - Une autre famille d'attaques dénommée *side channel attacks*<sup>3</sup> agissant directement sur l'implémentation de l'algorithme permet d'extraire des informations d'intérêt cryptographique lors de l'exécution de l'encryption.
- 
- Récemment (2015) une attaque de type *Meet in the Middle* (page 89) basé sur des structures bi-cycliques<sup>1 2</sup> a montré qu'il était possible de réduire l'effort nécessaire pour trouver une clé AES-128 à  $2^{126}$ , soit un facteur 4 par rapport au brute force. Ceci reste tout de même largement au dessus des capacités de calcul actuelles.
  - Une autre famille d'attaques dénommée *side channel attacks*<sup>3</sup> agissant directement sur l'implémentation de l'algorithme permet d'extraire des informations d'intérêt cryptographique lors de l'exécution de l'encryption.
  - La sécurité de AES (comme pour tout autre algorithme d'encryption) se base toujours sur l'hypothèse d'une clé d'entropie maximale. Les attaques publiées récemment sur des protocoles basés sur AES (comme WPA2) exploitent la faiblesse des passwords/passphrases qui sont à l'origine des clés utilisées par l'algorithme.

## Cryptanalyse Différentielle<sup>1</sup>

- Il s'agit d'une attaque *chosen plaintext* qui s'intéresse à la propagation des différences dans deux *plaintexts* au fur et à mesure qu'ils évoluent dans les différentes étapes de l'algorithme.
- Il attribue des probabilités aux clés qu'il "devine" en fonction des changements qu'elles induisent sur les *ciphertexts*. La clé la plus probable a des bonnes chances d'être la clé correcte après un grand nombre de couples *plaintext/ciphertext*.
- Nécessite  $2^{47}$  couples *chosen plaintext* pour obtenir des résultats corrects.

## Cryptanalyse Linéaire<sup>2</sup>

- Il s'agit d'une attaque *known plaintext* qui crée un simulateur du bloc à partir des approximations linéaires. En analysant un grand nombre de paires *plaintext/ciphertexts*, les bits de la clé du simulateur ont tendance à coïncider avec ceux du *block cipher* analysés (calcul probabiliste)
- Pour DES une attaque basée sur cette technique nécessite  $2^{38}$  *known plaintexts* pour obtenir une probabilité de 10% de deviner juste et  $2^{43}$  pour un 85%!
- Il s'agit de l'attaque analytique la plus puissante à ce jour sur les *block ciphers*.
- La mise en pratique des attaques différentielles et linéaires présente des difficultés dans la parallélisation des calculs par rapport à une recherche exhaustive de la clé.
- Ces deux attaques sont très sensibles au nombre d'étapes du *block cipher*: les chances de réussite augmentent exponentiellement au fur et à mesure que le nombre d'étapes de l'algorithme diminue.
- Une conjecture très répandue parmi les cryptographes est que ces attaques, à l'époque inédites, étaient connues par les concepteurs des DES. En particulier, le design des S-boxes offre une résistance très grande aux deux techniques.

## Attaque Meet-in-the-Middle

- S'applique aux constructions du type  $y := E_{K_2}(E_{K_1}(x))$ . L'espace de clés pour cette solution est de  $\{0,1\}^{112}$ . On construit d'abord deux listes  $L_1$  et  $L_2$  de  $2^{56}$  messages de la forme:  $L_1 = E_{K_1}(x)$  et  $L_2 = D_{K_2}(y)$  avec E et D les opérations d'encryption et decryption respectivement. Il faut alors identifier des éléments qui se répètent dans les deux listes et vérifier notre hypothèse avec un deuxième *known plaintext*. Les  $K_1$  et  $K_2$  associées à cette paire de *known plaintexts* seront (en toute vraisemblance) les clés recherchées!
- Effort nécessaire à réaliser les attaques:  $2^{57}$  opérations pour établir les deux listes +  $2^{56}$  blocs de 64 bits de stockage pour mémoriser les résultats intermédiaires... nettement inférieur au  $2^{112}$  estimé intuitivement...
- Ces techniques *meet-in-the-middle* sont aussi appliquées à la cryptanalyse interne des *block ciphers*.

**Fondement mathematique :**

- **Theorème Fondamental de l'Arithmétique:**

Tout nombre entier strictement positif  $n$  s'écrit de façon unique (à l'ordre près) comme un produit de puissances de nombres premiers  $p_i$  distincts, à savoir:

$$n = p_1^{e1} \cdot p_2^{e2} \cdot p_3^{e3} \cdot \dots \cdot p_m^{em}$$

- **Fonction Phi d'Euler:**

Soit  $n \in \mathbb{Z}^+$ , la **fonction phi d'Euler**  $\Phi(n)$  est égale au nombre de d'entiers positifs plus petits que  $n$  qui sont relativement premiers à  $n$ .

- **Calcul de la valeur de la fonction phi d'Euler  $\Phi(n)$ :**

D'après le théorème fondamental de l'arithmétique, tout nombre entier  $n > 1$  s'écrit:

$$n = \prod_{i=1}^m p_i^{ei}$$

alors  $\Phi(n)$  se calcule:

$$\Phi(n) = \prod_{i=1}^m (p_i^{ei} - p_i^{(ei-1)})$$

- **Théorème d'Euler:**

Soient  $n \in \mathbb{Z}^+$  et  $a \in \mathbb{Z}$  avec  $\text{pgcd}(a, n) = 1$ , alors on a:

$$a^{\Phi(n)} \equiv 1 \pmod{n}$$

- **Petit Théorème de Fermat** (cas particulier du théorème d'Euler si  $n$  est premier):

Soient  $a \in \mathbb{Z}$  et  $p$  un nombre premier tel que  $p$  ne divise pas  $a$ , alors on a:

$$a^{p-1} \equiv 1 \pmod{p}$$

A noter que puisque  $p$  est premier, on a  $\Phi(p) = p-1$ .

- **Réduction des exposants mod  $\Phi(n)$ :**

Si  $n$  est le produit de premiers distincts et  $r, s \in \mathbb{Z}$  t.q.  $r \equiv s \pmod{\Phi(n)}$  alors  $\forall a \in \mathbb{Z}$ :

$$a^r \equiv a^s \pmod{n}$$

- **Application du Théorème d'Euler au calcul des inverses:**

Suite au théorème d'Euler, on a que:

$$a a^{\Phi(n)-1} \equiv 1 \pmod{n}$$

ce qui signifie que  $a^{\Phi(n)-1}$  est l'inverse de  $a \pmod{n}$ . En particulier,  $a^{p-2}$  est l'inverse de  $a \pmod{n}$  si  $p$  est premier.

- **Définition:** Le groupe multiplicatif de  $Z_n$ , noté  $Z_n^*$  est = {  $a \in Z_n$  t.q.  $\text{pgcd}(a,n) = 1$  }.

En particulier, si  $n$  est premier:  $Z_n^* = \{ a \text{ t.q. } 1 \leq a \leq n-1 \}$

- Le **nombre d'éléments ou ordre du groupe multiplicatif**  $Z_n^*$  est  $\Phi(n)$  (par déf. de  $\Phi$ ).

- **Définition:** Soit  $a \in Z_n$ , l'**ordre** de  $a$  est le plus petit entier positif  $t$  pour lequel:

$$a^t \equiv 1 \pmod{n}$$

- **Définition:** Soit  $\alpha \in Z_n^*$ , si l'ordre de  $\alpha$  est  $\Phi(n)$ , alors  $\alpha$  est un **générateur** de  $Z_n^*$ .

Lorsqu'un groupe  $Z_n^*$  a un générateur, on dit qu'il est **cyclique**.

- **Propriétés des générateurs:**

- $Z_n^*$  a un générateur ssi.  $n = 2, 4, p^k$  ou  $2p^k$ , avec  $p$  premier,  $p \neq 2$  et  $k \geq 1$ .

En particulier, si  $p$  est premier,  $Z_p^*$  a un générateur.

- Si  $\alpha$  est un générateur de  $Z_n^*$ , alors tous les éléments de  $Z_n^*$  s'écrivent:

$$Z_n^* = \{ \alpha^i \pmod{p} \text{ t.q. } 0 \leq i \leq \Phi(n) - 1 \}$$

- Le nombre de générateurs de  $Z_n^*$  est  $\Phi(\Phi(n))$ .

- $\alpha$  est un générateur de  $Z_n^*$  ssi.  $\forall$  premier  $p$  t.q.  $p$  divise  $\Phi(n)$ , on a:

$$\alpha^{\Phi(n)/p} \neq 1 \pmod{n}.$$

En particulier si  $n$  est un **premier** de la forme  $2p + 1$  avec  $p$  premier (un tel  $n$  est appelé un **safe prime**),  $\alpha$  est générateur de  $Z_n^*$  ssi.  $\alpha^2 \neq 1 \pmod{n}$  et  $\alpha^p \neq 1 \pmod{n}$ .

## Fast Exponentiation

- **Fast exponentiation:** En utilisant la représentation binaire d'un nombre, on peut calculer des puissances très efficacement, p.ex calcul de  $2^{644} \pmod{645}$ :

$$(644)_{10} = (1010000100)_2$$

Maintenant, on calcule les exposants correspondants aux puissances de 2, soient:

$$2 \equiv 2 \pmod{645}$$

$$2^2 \equiv 4 \pmod{645}$$

$$2^4 \equiv 16 \pmod{645}$$

$$2^8 \equiv 256 \pmod{645}$$

$$2^{16} \equiv 391 \pmod{645}$$

...

$$2^{512} \equiv 256 \pmod{645}$$

D'après la représentation binaire, on calcule:

$$2^{644} = 2^{512+128+4} = 2^{512}2^{128}2^4 = 1601536 \equiv 1 \pmod{645}$$

La complexité de cet algorithme *fast exponentiation* est **O ( $\log^3 n$ )**.

En s'appuyant sur le théorème d'Euler, le calcul de l'inverse d'un nombre dans un tel groupe est donc effectué en temps polynomial.

- **L'algorithme d'Euclide étendu** peut être également utilisé pour trouver un  $x$  t.q:

$ax \equiv 1 \pmod{n}$  puisque cette congruence s'écrit:  $ax - 1 = kn$  et donc:

$ax - kn = 1 = \text{pgcd}(a, n)$ . La complexité de cet algorithme est également **O ( $\log^3 n$ )**.

RSA :

### Génération des clés

- Chaque entité (A) crée une paire de clés (publique et privée) comme suit:
  - A choisit la taille du **modulus n** (p.ex. taille (n) = 1024 ou taille (n) = 2048).
  - A génère deux nombres premiers **p** et **q** de grande taille ( $\sim n/2$ ).
  - A calcule  $n := pq$  et  $\Phi(n) = (p-1)(q-1)$ .
  - A génère l'exposant d'encryption **e**, avec  $1 < e < \Phi(n)$  t.q.  $\text{pgcd}(e, \Phi(n)) = 1$ .
  - A calcule l'exposant de decryption **d**, t.q.:  $ed \equiv 1 \pmod{\Phi(n)}$  avec l'algorithme d'Euclide étendu ou avec l'algorithme *fast exponentiation* (page 94).
  - Le couple  $(n, e)$  est la **clé publique de A**; **d** est la **clé privée de A**.

### Encryption

- L'entité **B** obtient  $(n, e)$ , la clé publique authentique de **A**.
- **B** transforme son plaintext en une série d'entiers  $m_i$ , t.q.  $m_i \in [0, n-1] \forall i$ .
- **B** calcule le ciphertext  $c_i := m_i^e \pmod{n}$ ,  $\forall i$  avec l'algorithme *fast exponentiation*.
- **B** envoie à **A** tous les ciphertext  $c_i$ .

### Decryption

- **A** utilise sa clé privée pour calculer les plaintexts  $m_i = c_i^d \pmod{n}$ .

---

El gamal :

### Génération des clés

- Chaque entité (A) crée une paire de clés (publique et privée) comme suit:
  - A génère un nombre premier **p** de grande taille ( $\text{len}(p) \geq 1024$  bits) et un générateur  $\alpha$  du groupe multiplicatif  $Z_p^*$ .
  - A génère un nombre aléatoire **a**, t.q.  $1 \leq a \leq p-2$  et calcule  $\alpha^a \pmod{p}$ .
  - La clé publique de A est  $(p, \alpha, \alpha^a \pmod{p})$ , la clé privée de A est **a**.

### Encryption

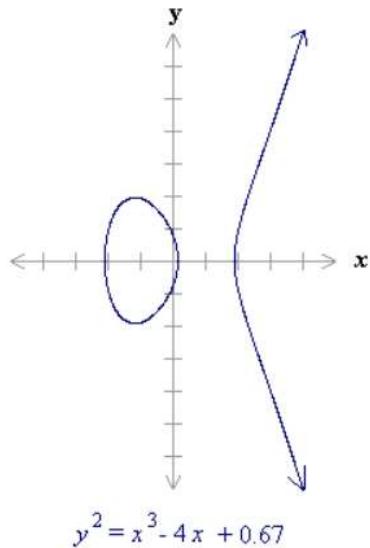
- L'entité **B** obtient  $(p, \alpha, \alpha^a \pmod{p})$ , la clé publique authentique de **A**.
- **B** transforme son plaintext en une série d'entiers  $m_i$ , t.q.  $m_i \in [0, p-1] \forall i$ .
- Pour chaque message  $m_i$ :
  - **B** génère un nombre aléatoire unique **k**, t.q.  $1 \leq k \leq p-2$ .
  - **B** calcule  $\lambda := \alpha^k \pmod{p}$  et  $\delta := m_i (\alpha^a)^k \pmod{p}$  et envoie le ciphertext  $c := (\lambda, \delta)$ .

### Decryption

- **A** utilise sa clé privée **a** pour calculer  $\lambda^{p-1-a} \pmod{p}$  (à noter que:  
 $\lambda^{p-1-a} \equiv \lambda^{-a} \equiv \alpha^{-ak} \pmod{p}$ ).
- **A** retrouve le plaintext en calculant:  $(\alpha^{-ak}) \delta \pmod{p}$ .

### Courbe elliptique :

- Une **courbe elliptique** est un ensemble de points  $E$  défini par l'équation:  
 $y^2 = x^3 + ax + b$ , avec  $x, y, a$  et  $b$  des nombres **rationnels**, **entiers** ou **entiers modulo  $m$**  ( $m > 1$ ). L'ensemble  $E$  contient également un "point à l'infini" noté  $\infty$ . Le point  $\infty$  n'est pas dans la courbe mais il est l'élément identité de  $E$ .
- On choisira pour nos calculs les courbes elliptiques n'ayant pas de racines multiples ou, en d'autres termes, des courbes où le discriminant  $4a^3 + 27b^2 \neq 0$ .



- L'avantage principal de la cryptographie publique basée sur des courbes elliptiques est que la taille des nombres utilisés (et donc, des clés) est plus petite.
- Ceci est dû à la complexité accrue des calculs sur  $E_p$  (courbe elliptique définie sur le corps  $Z_p$ ) par rapport aux corps habituels tels que  $Z_p$  ou  $GF(2^m)$ .
- Ce tableau montre les rapports des tailles des clés par rapport à celles de RSA:

| NIST guidelines for public key sizes for AES |                        |                   |                        |
|----------------------------------------------|------------------------|-------------------|------------------------|
| ECC KEY SIZE<br>(Bits)                       | RSA KEY SIZE<br>(Bits) | KEY SIZE<br>RATIO | AES KEY SIZE<br>(Bits) |
| 163                                          | 1024                   | 1:6               |                        |
| 256                                          | 3072                   | 1:12              | 128                    |
| 384                                          | 7680                   | 1:20              | 192                    |
| 512                                          | 15 360                 | 1:30              | 256                    |

Supplied by NIST to ANSI X9.61

- La représentation d'un plaintext en points de la courbe reste une opération complexe.
- En Octobre 2003<sup>1</sup>, la *US National Security Agency* (NSA) a acheté un brevet de *Certicom* pour l'utilisation de la cryptographie à courbes elliptiques.
- En Septembre 2013 Claus Diem montré<sup>2</sup> que sous certaines conditions le problème **ECDLP** pouvait être résolu en temps *sub-exponentiel*.

# Procédé d'ElGamal sur des Courbes Elliptiques

## Génération des clés

- Chaque entité (A) crée une paire de clés (publique et privée) comme suit:
  - A choisit une courbe elliptique  $E_p$  avec  $p$ , un nombre premier de grande taille ( $\text{len}(p) \sim 200 \text{ bits}$ ) et un point  $P_0 \in E_p$ .
  - A génère un nombre aléatoire  $a$ , t.q.  $1 < a < p$  et calcule  $P_a = aP_0$  (multiplication par un scalaire sur  $E_p$ , pour laquelle, il existe des algorithmes efficaces).
  - La clé publique de A est  $(E_p, P_0, P_a)$ , la clé privée de A est  $a$ .

## Encryption

- L'entité B obtient  $(E_p, P_0, P_a)$ , la clé publique authentique de A.
- B transforme son plaintext en une série d'entiers  $m_i$ , t.q.  $m_i \in E_p \forall i$ .
- Pour chaque message  $m_i$ :
  - B génère un nombre aléatoire unique  $k$ , t.q.  $1 < k < p$ .
  - B calcule  $\lambda := kP_0$  et  $\delta := kP_a + m_i$  et envoie le ciphertext  $c := (\lambda, \delta)$ .

## Decryption

- A utilise sa clé privée  $a$  pour calculer:  $a\lambda = akP_0 = kP_a$ .
- A retrouve le plaintext en calculant:  $\delta - kP_a = kP_a + m_i - kP_a = m_i$ .
- Le sécurité du schéma s'appuie sur ECDLP!

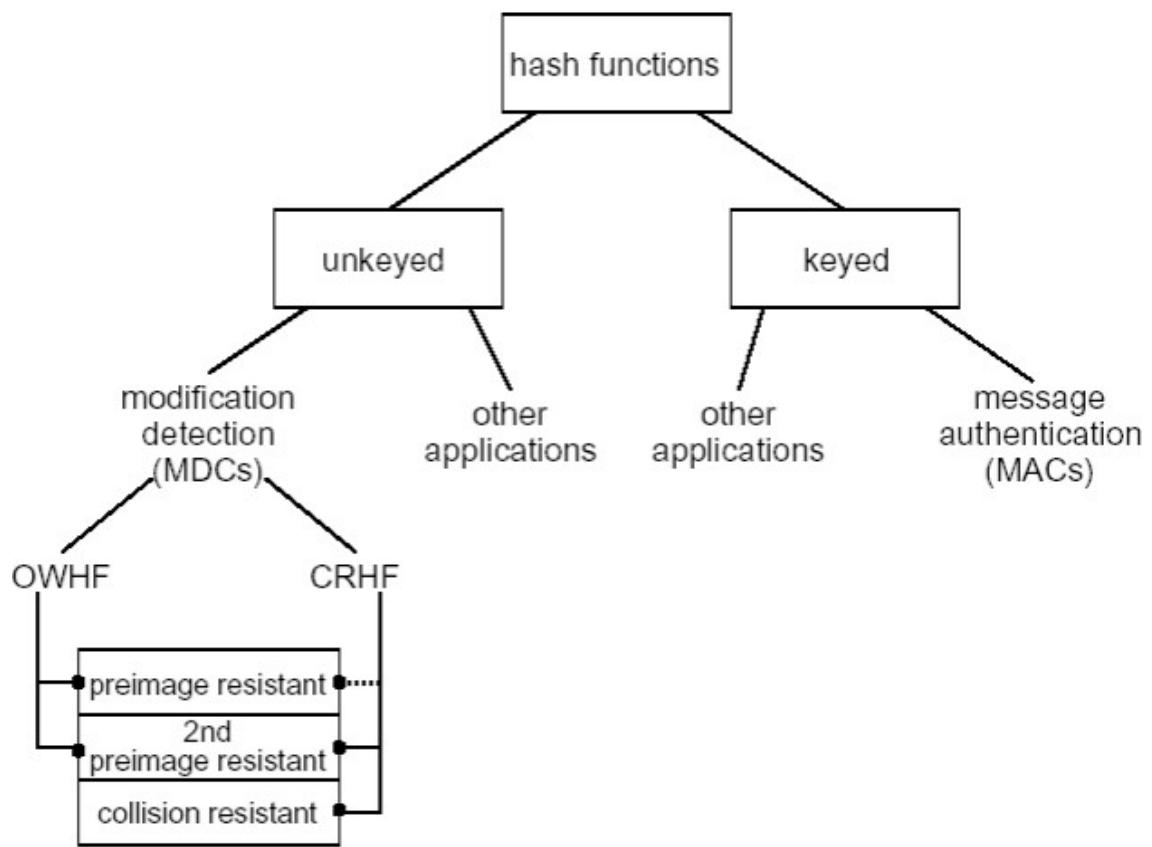
## Hachage:

- Une fonction de hachage (**hash function**) est une fonction  $h$  ayant les propriétés suivantes:
  - **compression**: la fonction  $h$  fait correspondre à un ensemble  $X$  composée par des chaînes de bits **de longueur finie mais arbitraire**, un ensemble  $Y$  composé par des chaînes de bits **de longueur finie et fixée** (et normalement inférieur à la taille de  $X$ ) avec  $h(x) = y$ , et  $x \in X, y \in Y$ .
  - **facile à calculer**: partant de  $h$  et  $x \in X$ ,  $h(x)$  est facile à calculer.
- Une hash function est dite “à clé” (**keyed hash function**) si une clé intervient dans le calcul de la fct. ( $h_k(x) = y$ ); sinon on l'appelle “sans clé” (**unkeyed hash function**).
- Les hash functions ont des nombreuses applications informatiques dont l'archivage structuré facilitant la recherche. Coté sécurité nous allons étudier deux catégories principales:
  - codes détecteurs d'altérations (**manipulation detection codes (MDC)** or **message integrity codes (MIC)**): ce sont des *unkeyed functions* permettant de fournir un service d'intégrité sous certaines conditions. Le résultat d'une telle fonction est appelée *MDC-value* ou, simplement, *digest*.
  - codes d'authentification de message (**message authentication codes ou MAC**) qui sont des *keyed functions* permettant d'authentifier la source du message et d'assurer son intégrité sans utiliser des mécanismes (cryptage) additionnels.

- Quelques propriétés de base des hash functions:
  - 1) *preimage resistance*: étant donné un  $y \in Y$ , il est calculatoirement impossible de trouver une pré-image  $x \in X$  satisfaisant  $h(x) = y$ .
  - 2) *2<sup>nd</sup>-preimage resistance*: étant donné un  $x \in X$  et son image  $y \in Y$ , avec  $h(x) = y$ , il est calculatoirement impossible de trouver un  $x' \neq x$  tel que  $h(x) = h(x')$ . Aussi appelée *weak collision resistance*.
  - 3) *collision résistance*: il est calculatoirement impossible de trouver deux pré-images  $x, x' \in X$  distinctes pour lesquels  $h(x) = h(x')$  (pas de restriction sur le choix des valeurs). Aussi appelée *strong collision resistance*.
- Une fonction de hachage à sens unique (**one way hash function ou OWHF**) est un MDC satisfaisant 1) et 2). Aussi appelée: *weak one-way hash function*.
- Une fonction de hachage résistante aux collisions (**collision resistant hash function ou CRHF**) est un MDC satisfaisant le propriétés 2) et 3). (A noter que 3)  $\Rightarrow$  2)). Aussi appelée: *strong one-way hash function*.
- OWF  $\not\Rightarrow$  OWHF: A noter qu'une OWHF en tant que hash function impose des restrictions supplémentaires sur les domaines sources et image ainsi que sur la 2nd-*preimage resistance* qui ne sont pas forcement respectés par des OWFs.
  - Exemple:  $f(x) = x^2 \bmod n$  avec  $n = pq$  ( $p$  et  $q$  inconnus) n'est pas une OWHF car étant donné  $x$ ,  $-x$  est une collision triviale.

**MAC :**

- Un **Message Authentication Code (MAC)** est une famille de fonctions  $h_k$  paramétrisées par une clé secrète  $k$  ayant les propriétés suivantes:
  - 1) **compression**: comme pour les fonctions de hash génériques mais appliqué à  $h_k$ .
  - 2) **facile à calculer**: à partir d'une fonction  $h_k$ , et d'une clé connue  $k$ , on peut facilement calculer  $h_k(x)$ . Le résultat est appelée un *MAC-value* ou, simplement, un *MAC*.
  - 3) **résistance calculatoire (computation-resistance)**: sans connaissance de la clé symétrique  $k$ , il est (calculatoirement) impossible de calculer des paires  $(x, h_k(x))$  à partir de 0 ou plusieurs paires connus  $(x_i, h_k(x_i))$  pour tout  $x \neq x_i$ .
- La propriété 3) implique que les paires  $(x_i, h_k(x_i))$  ne peuvent non plus servir à calculer la clé  $k$  (*key non-recovery*). Cependant la propriété *key non-recovery* n'implique pas *computation-resistance* car des attaques *chosen/known -plaintext* pourraient mener à des paires  $(x, h_k(x))$  falsifiées.
- L'impossibilité de calculer des paires  $(x, h_k(x))$  se traduit également en *preimage* et *collision resistance* (cf. transparent précédent) pour toute entité ne possédant pas la clé  $k$ .



## Résistance Calculatoire Théorique des Hash Functions: Récapitulation

| Type de Hash Fct. | Caractéristique                                                         | Difficulté Calculatoire   | But de l'attaque                                          | Taille conseillée du digest/clé |
|-------------------|-------------------------------------------------------------------------|---------------------------|-----------------------------------------------------------|---------------------------------|
| OWHF              | <i>preimage resistance</i><br><i>2<sup>nd</sup>-preimage résistance</i> | $2^n$<br>$2^{n-1}$        | trouver une pré-image<br>trouver $x'$ avec $h(x') = h(x)$ | $n \geq 80$ bits                |
| CRHF              | <i>collision resistance</i>                                             | $2^{n/2}$                 | trouver une collision                                     | $n \geq 160$ bits               |
| MAC               | <i>key non-recovery computation resistance</i>                          | $2^t$<br>$\min(2^t, 2^n)$ | trouver la clé<br>produire un $(x, h_k(x))$               | $t \geq 80$                     |

- n: taille du *MDC-value* ou du *MAC-value* résultant de l'application de la *hash function*
- t: taille de la clé du MAC

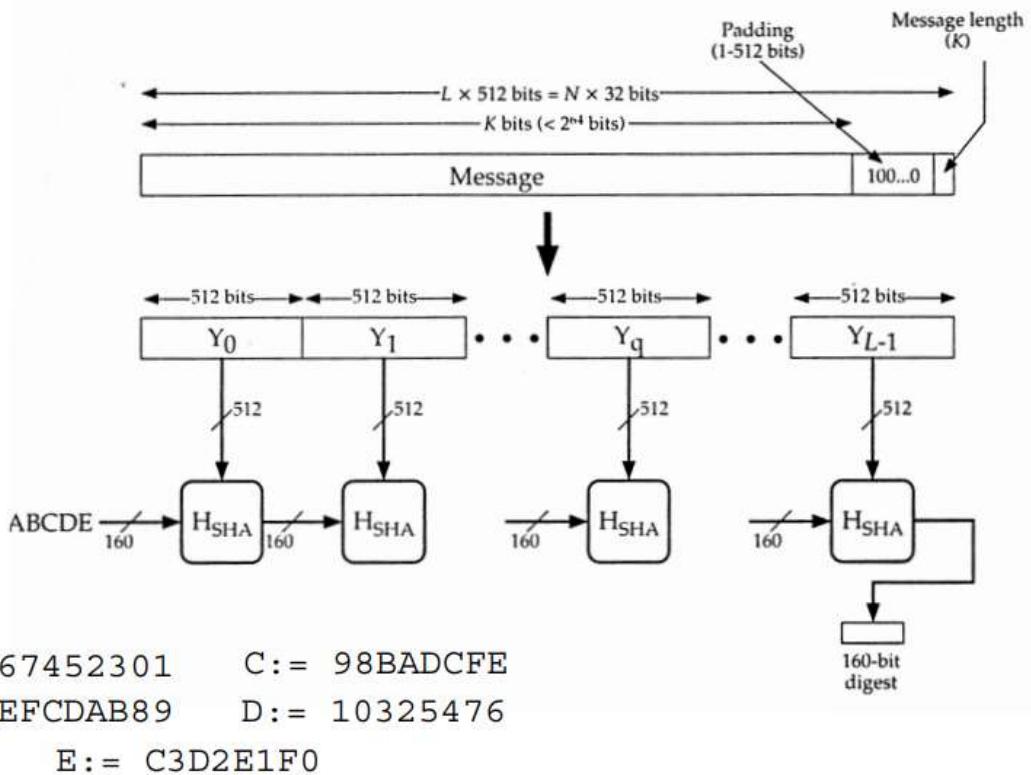
Si  $k$  est grand, on remplace  $k(k-1)$  par  $k^2$  et on obtient après des calculs simples:

$$k = \sqrt{2(\ln 2)n} \approx 1,17\sqrt{n} \approx \sqrt{n}$$

- En prenant  $n = 365$  pour l'anniversaire, on obtient  $k = 22.3$ , ce qui confirme l'énoncé du problème

## MDCs Basées sur des Systèmes de Cryptage

- Idée: utiliser un système de cryptage symétrique connu pour construire un MDC.
- Problèmes à résoudre:
  - il faut “casser” la réversibilité des algorithmes symétriques pour en faire des OWHF ou des CRHF.
  - La “largeur nominale” de certains systèmes de cryptage (eg. DES) est de 64 bits, ce qui n'est pas suffisant pour construire des CRHF.
- Principe de fonctionnement:
  - les blocs de texte sont séquentiellement traités par la “boîte” de cryptage.
  - la compression se base sur des opérations de chaînage avec les blocs résultant des itérations précédentes et des fonctions logiques (fondamentalement XOR). Ceci rend également le procédé irréversible.
  - Si nécessaire,  $n$  boîtes de cryptage seront combinées pour obtenir des longueurs de *digests*  $n$  fois supérieures à la largeur nominale des boîtes utilisées.
- Attention: la sécurité de ces algorithmes est fortement dépendante des propriétés des boîtes de cryptage sous-jacents.
- Il s'agit de fonctions conçues exclusivement pour générer des codes d'intégrité (des *digests*) avec un souci principal de vitesse et sécurité.
- Leur fonctionnement se base sur les éléments suivants:
  - des opérations d'initialisation (*padding* + rajouter la longueur).
  - un ensemble de constantes prédéfinies choisies spécialement pour augmenter la dispersion.
  - un ensemble “d'étapes” (*rounds*) qui vont séquentiellement s'appliquer à tous les blocs des données originaux. Ces rounds vont effectuer une combinaison d'opérations logiques et des rotations sur les données et les constantes.
  - des opérations de chaînage impliquant les sorties des *rounds* précédents.
- Dans ces fonctions, chaque bit du *digest* est une fonction de chaque bit des entrées.
- Les plus connues sont:
  - MD5: R. Rivest, 1992; RFC 1321. *Digest = 128 bits. Cassé!*
  - SHA-0: NIST, 1993. *Digest = 160 bits. Collisions en  $2^{39}$  opérations au lieu de  $2^{80}$*
  - SHA-1: NIST, 1995. *Digest = 160 bits. Révision de SHA-0 avec rotation de bits additionnelle. Collisions en  $2^{63}$  opérations (au lieu de  $2^{80}$ ).*
  - SHA-2: NIST (FIPS 190-3). Comprend: SHA-224, SHA-256, SHA-384 et SHA-512. Les tailles du digest vont de 224 à 512 bits.
  - SHA-3: Keccak Algorithm (taille du digest variable de 224 à 512 bits)



### Nested MACs et HMACs :

- Un **Nested MAC** ou **NMAC** est une composition de 2 familles de fonctions MACs G et H paramétrées par les clés k et l tel que:  

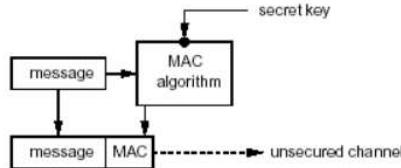
$$G \circ H = \{ g \circ h \text{ avec } g \in G \text{ et } h \in H \} \text{ avec } g \circ h_{(k,l)}(x) = g_k(h_l(x))$$
- La sécurité d'un NMAC dépend de deux critères:
  - La famille de fonctions G est résistante aux collisions.
  - La famille de fonctions H est résistante aux attaques spécifiques pour MACs, i.e.:  
 Il est impossible de trouver un couple  $(x,y)$  et une clé m fixée mais inconnue, telle que:  $\text{MAC}_m(x) = y$ .
- Un **HMAC** (FIPS 198, 2002) est un Nested MAC utilisant à la base des MDCs sans clé dédiées comme SHA-1 ou SHA-256.
- Un HMAC utilise deux constantes de 512 bits dénommées *ipad* et *opad* telles que:  

$$\text{opad} := 363636 \dots 36 \quad \text{et} \quad \text{ipad} := 5C5C5C \dots 5C$$
 et une clé k de 512 bits.
  - Le schéma de fonctionnement de HMAC-256 (sur la base de SHA-256) est le suivant:  

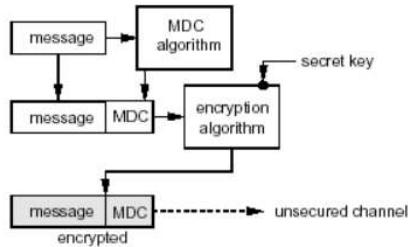
$$\text{HMAC-256}_k(x) := \text{SHA-256}((k \oplus \text{opad}) \parallel \text{SHA-256}((k \oplus \text{ipad}) \parallel x))$$
  - Les HMACs sont les MACs les plus utilisés. Les attaques mentionnées sur les fonctions de la famille SHA sont plus difficiles à réaliser sur un HMAC par cause de la clé k.

## Applications des Hash Functions: Intégrité

### MAC Seul:

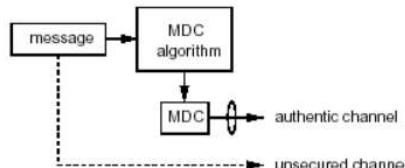


### MDC + Encryption:



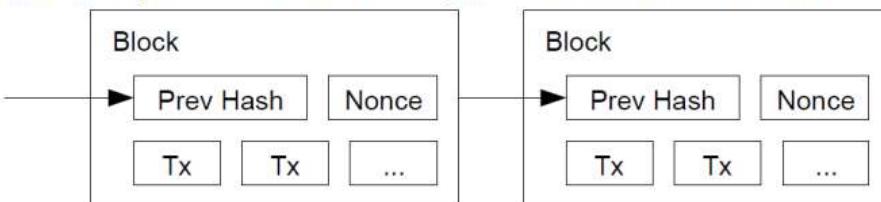
### MDC + canal authentique:

(Source [Men97])



## Application des Hash Functions: Blockchains

- Les transactions bitcoin sont publiées et visibles par tous les intervenants. Elles sont encapsulées dans des **blocs** chaînés à l'aide de fonctions de hachage cryptographiques
- Le **minage (mining)** consiste à rajouter itérativement des nouveaux blocs contenant les transactions courantes
- La génération d'un bloc valable nécessite la **Résolution d'un puzzle cryptographique (proof of work)** très coûteux en temps de calcul (trouver des *pseudo-collisions* dans les fonctions de hachage cryptographiques). La validation reste très efficace
- Le premier mineur capable de générer un bloc valable recevra une récompense monétaire (en bitcoins). Le processus de minage est ouvert à tous les mineurs mais seul le premier est récompensé
- La chaîne de blocs résultante (**blockchain**) devient alors un **registre public (public ledger)**, décentralisé et immuable protégeant toutes les transactions passées. La falsification/modification des données protégées par la *blockchain* nécessiterait un effort calculatoire supérieur à celui effectué par tous les mineurs *honnêtes*



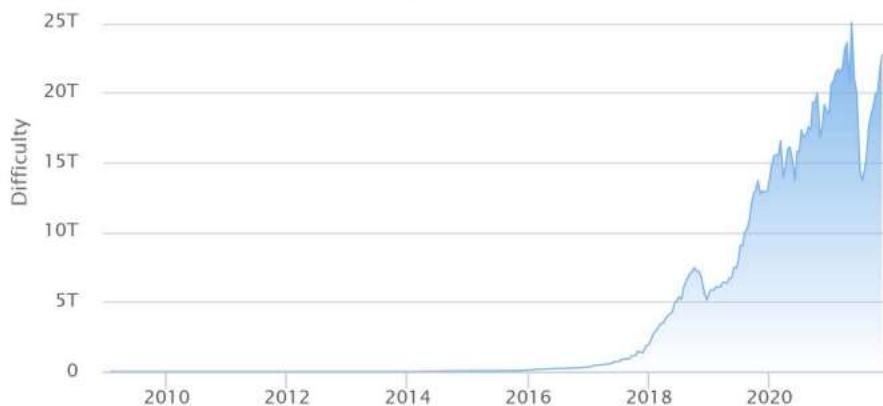
Source Image: Bitcoin: A Peer-to-Peer Electronic Cash System. Satoshi Nakamoto

---

## Blockchain: *Proof of Work*

**Statistiques Bitcoin 20/11/2021 (source <http://btc.com>):**

- **Difficulty:** 22,674,148,233,453
- **Target:**  $2^{224} / \text{Difficulty} = \sim 2^{180}$ . Le digest valable pour générer un bloc doit être inférieur à  $2^{180}$ , ce qui signifie une **pseudo-collision sur les 76 bits de poids plus fort**. La variation sur les inputs dépend du *nonce*
- **Hashrate:**  $\sim 168 \text{ EH/sec}$  ( $168 * 10^{18} \text{ hashes/sec}$ )
- Fonctions de hachage exécutées pour obtenir un bloc:  $\sim 97 * 10^{21}$
- Temps de génération d'un bloc: **9,5 minutes**



## Autres Applications de Hash Functions

- Authentification:
  - *data origin authentication* (DOA)
  - *transaction authentication* (= DOA + *time-variant parameters*)
- *Virus checking*
  - Le créateur d'un logiciel crée un digest =  $h(x)$  avec  $x$  étant l'original et le distribue par un canal sûr (eg. CD-ROM).
- Distribution des clés publiques
  - Permet de contrôler l'authenticité d'une clé publique.
- *Timestamp* sur un document:
  - Le document sur lequel on veut effectuer le timestamp est d'abord soumis à une hash function. Le timestamp (avec la signature de l'entité correspondante) s'applique alors seulement au digest.
- *One-time password (S-Key)* (mécanisme d'identification)
  - A partir d'un *seed* secret  $x_0$ , on crée une chaîne de hash-values:  $x_1 = h(x_0)$ ,  $x_2 = h(x_1)$ , ...  $x_n = h(x_{n-1})$ .
  - Le système mémorise  $x_n$  et l'utilisateur rentre  $x_{n-1}$ . Si  $h(x_{n-1}) == x_n \Rightarrow \text{OK}$ .
  - Le système mémorise alors  $x_{n-1}$  et ainsi de suite.

## *Randomized Hash Functions* (l'exemple UNIX) I

- UNIX garde ses mots de passe dans un fichier globalement accessible (ou éventuellement distribué par NIS)
- L'information stockée correspond au résultat produit par une hash function.
- Exemple (fictif):

```
root:Jw87u9bebeb9i:0:1:Operator:/bin/csh
pp:1Qhw.oihEtHK6:359:355:PP:/net/spp_telecom/pp:/bin/cs
```

- Problèmes:

- la hash function étant déterministe, elle produit le même résultat pour des mots de passe identiques.
- on pourrait créer des “cahiers” (*codebooks*) contenant le résultat de l'application de la hash function à des entrées données (p.ex. un dictionnaire) et les comparer facilement (*off-line*) avec les chaînes stockées par UNIX (*brute force dictionary attack*).

- Solution:

- Rajouter un élément (pseudo) aléatoire de 12 bits différent pour chaque mot de passe (appelé *salt*) avant de calculer la hash function et lors de la vérification.
- Cet élément permet de rajouter un facteur aléatoire de 4096 possibilités pour chaque mot de passe et de prévenir la détection des duplications.

### **Signature digitale :**

- *Signature digitale*: chaîne de données permettant d'associer un message (sous forme digitale) à une entité d'origine.
- *Schéma de signature digitale*: algorithme de génération + algorithme de vérification.
- *Procédé de signature*: formatage du message + algorithme de génération de signature
- *Procédé de vérification*: algorithme de vérification + (reconstruction du message).
- Classification des signatures digitales:
  - *Signatures digitales avec appendice* qui nécessitent la présence du message original pour vérifier la validité de la signature. Ce sont les plus couramment utilisées.  
Exemples: *ElGamal, DSS*.
  - *Signatures digitales avec reconstitution du message* qui offrent, en plus, la possibilité de reconstruire le message à partir de la signature. Exemples: *RSA, Rabin*.
- Les signatures digitales sont pour la plupart basées sur la crypto asymétrique du fait que la notion clé partagée n'est pas adaptée aux besoins d'identifier une entité de façon explicite.
- Des engagements semblables à ceux obtenus par une signature à clé publique (comme la non-répudiation d'origine) peuvent cependant être obtenus avec la technologie symétrique et des tierces de confiance (*Trusted Third Parties* ou *TTP*). Ces méthodes sont nommées: *arbitrated digital signatures*.

## Signatures Digitales avec Appendice: Cadre Formel

- On admet que chaque entité a une clé privée pour signer des messages et une copie authentique des clés publiques des correspondants.

Notation:  $M$ : Espace de messages

$M_h$ :  $m_h = H(m)$  avec  $m \in M$ ,  $m_h \in M_h$  et  $H$  une *hash function*

$S$ : Espace des valeurs pouvant être obtenues par un procédé de signature

Description:

Chaque entité définit une appl. injective  $S_A : M_h \rightarrow S$ ; (ie. la *signature*)

L'application  $S_A$  donne lieu à une application  $V_A$ :

$V_A : M_h \times S \rightarrow \{\text{vrai, faux}\}$ ; (ie. la *vérification*)

t.q.  $\forall m_h \in M_h, s \in S$ , on a:

$V_A(m_h, s) = \text{vrai si } S_A(m_h) = s \text{ et}$

$V_A(m_h, s) = \text{faux sinon}$

Les opérations  $S_A$  nécessitent la clé *privée* de A alors que les opérations  $V_A$  utilisent la clé *publique* de A.

- Quelques propriétés simples:

- Les opérations  $S_A$  et  $V_A$  doivent être faciles à calculer (en ayant les clés corresp.)

- Il est impossible (calculatoirement) pour une entité n'ayant pas la clé privée de A de trouver un  $m'$  et un  $s'$  avec  $m' \in M$  et  $s' \in S$  t.q.  $V_A(m'_h, s') = \text{vrai}$  avec  $m'_h = H(m')$ .

## SD avec reconstitution du message: Cadre Formel

Notation: en plus des définitions précédentes, on a:

$M_s$ : L'espace des éléments sur lesquels peut s'appliquer une signature.

$R$ : Une application injective:  $M \rightarrow M_s$ , appelée *fonction de redondance*. Elle doit être *inversible et publique*.

$M_R$ :  $M_R = \text{Im}(R)$

Description:

Chaque entité définit une appl. injective  $S_A : M_S \rightarrow S$ ; (ie. la *signature*)

L'application  $S_A$  donne lieu à une application  $V_A$ ; (ie. la *vérification*)

$V_A : S \rightarrow M_s$ , t.q.  $V_A \circ S_A = \text{Identité sur } M_s$

A noter que la vérification s'effectue sans la clé privée de A

Génération de signature:

(1) Calculer  $m_R = R(m)$  et  $s = S_A(m_R)$

(2) Rendre publique  $s$  en tant que signature de A sur  $m$ . Ceci permet aux autres entités de vérifier la signature et reconstituer  $m$ .

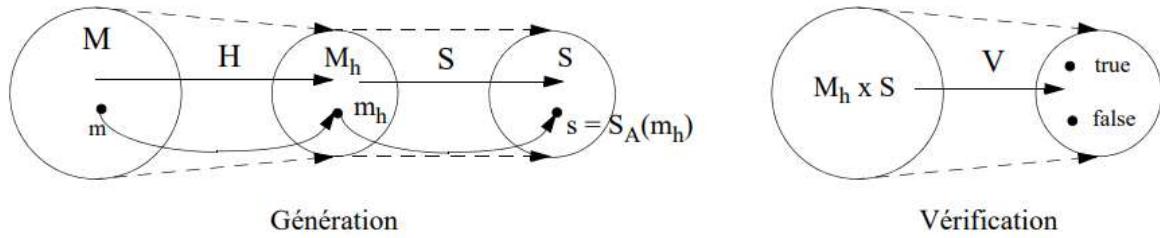
Vérification:

(1) Calculer  $m_R = V_A(s)$  (avec la clé publique de A)

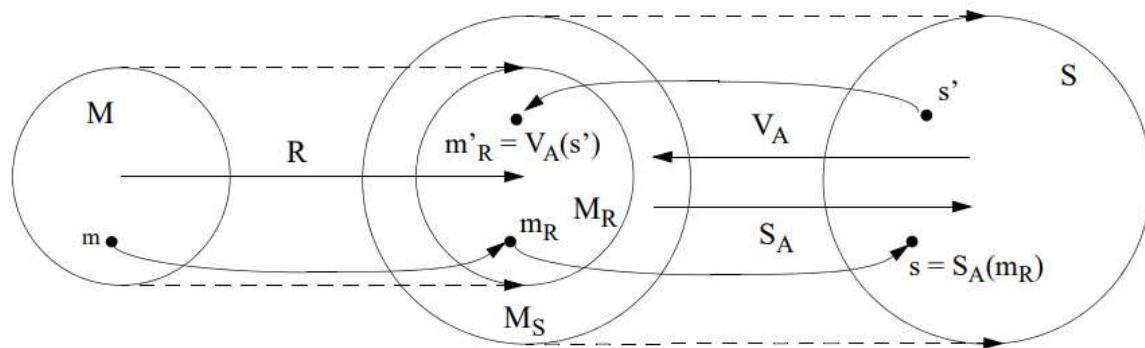
(2) Vérifier que  $m_R \in M_R$  (sinon rejeter la signature)

(3) Reconstituer  $m$  en calculant:  $R^{-1}(m_R)$

### Signature Digitale avec appendice



### Signature Digitale avec réconstitution du message



## SD avec reconstitution du message: Propriétés

- Propriétés:
  - Les opérations  $S_A$  et  $V_A$  doivent être faciles à calculer (en ayant les clés corresp.)
  - Il est impossible (calculatoirement) pour une entité n'ayant pas la clé privée de A de trouver un  $s' \in S$  t.q.  $V_A(s') \in M_R$
- Remarques sur la fonction de redondance:
  - Le choix d'une fonction de redondance est essentiel pour la sécurité du système.
  - Si  $M_R = M_S$  et  $R$  et  $S_A$  sont des bijections respectivement de  $M$  dans  $M_R$  et de  $M_S$  dans  $S$ , alors  $M$  et  $S$  ont une taille identique et, par conséquent, il est trivial de forger des messages portant la signature de A.
- Exemple de fonction de redondance: soit  $M = \{m: m \in \{0,1\}^n\}$  ( $n$  taille du message) et  $M_S = \{t: t \in \{0,1\}^{2n}\}$ . Soit  $R : M \rightarrow M_S$  t.q.  $R(m) = m \parallel m$  ( $\parallel$  étant la concaténation de 2 messages). La probabilité de tomber sur un tel message en essayant de forger un message à partir d'une signature est de :  $|M_R| / |M_S| = (1/2)^n$ , ce qui est négligeable pour des grands messages.
- Attention!: Une fonction de redondance adaptée pour un schéma de signature digitale peut provoquer des failles dans un autre différent !

**Signature RSA :**

## Génération des clés

- Chaque entité (A) crée une paire de clés (publique et privée) comme suit:
  - A choisit la taille du **modulus n** (p.ex. taille (n) = 1024 ou taille (n) = 2048).
  - A génère deux nombres premiers **p** et **q** de grande taille ( $\sim n/2$ ).
  - A calcule **n := pq** et  $\Phi(n) = (p-1)(q-1)$ .
  - A génère l'exposant de vérification **e**, avec  $1 < e < \Phi(n)$  t.q.  $\text{pgcd}(e, \Phi(n)) = 1$ .
  - A calcule l'exposant de signature **d**, t.q.:  $ed \equiv 1 \pmod{\Phi(n)}$  avec l'algorithme d'Euclide étendu ou avec l'algorithme *fast exponentiation* (page 94).
  - Le couple **(n,e)** est la **clé de publique de A**; **d** est la **clé privée de A**.

## Signature

- A calcule la fonction de redondance du message **m**:  $m_R := R(m)$ .
- A calcule la signature:  $s := m_R^d \pmod{n}$  et envoie **s** à **B**.

## Vérification

- L'entité **B** obtient **(n,e)**, la clé publique authentique de **A**.
- B calcule  $m'_R = s^e \pmod{n}$ , vérifie  $m'_R \in M_R$  et rejette la signature si  $m'_R \notin M_R$ .
- B retrouve le message correctement signé par **A** en calculant:  $m = R^{-1}(m'_R)$ .
- La preuve de fonctionnement est identique à celle du procédé d'encryption (page 99).  
L'ordre d'exponentiation n'a pas d'influence puisque:  
$$ed \equiv de \equiv 1 \pmod{\Phi(n)}$$
- Le procédé peut également être utilisé pour produire des signatures avec appendice avec les modifications suivantes:

### Signature:

- A utilise une fonction de hachage **H** et calcule  $m_h := H(m)$ .
- A calcule la signature de  $m_h$ :  $s := m_h^d \pmod{n}$  et envoie le couple **(m,s)** à **B**.

## Vérification

- B calcule  $m'_h = s^e \pmod{n}$  et  $H(m)$  et vérifie l'égalité  $m'_h = H(m)$ .
- Si l'égalité est vérifiée, **B** accepte la signature **s** de **A** sur le message **M**.
- Le calcul de signature est plus lent que la vérification à cause de différence de taille entre l'exposant **d** ( $\text{taille}(d) \approx \text{taille}(\Phi(n))$ ) et **e**.
- Les risques et attaques mentionnés dans le procédé d'encryption (page 102) s'appliquent également pour la signature.
- Il convient de différencier les paires de clés d'encryption et de signature puisqu'elles nécessitent des politiques de stockage, sauvegarde et mise à jour distinctes.

### Signature aveugle :

- Schéma inventé par Chaum ([Chau82]<sup>1</sup>).
- Idée: **A** envoie une information à **B** pour signature. **B** retourne à **A** l'information signée. A partir de cette signature, **A** peut calculer la signature de **B** sur un autre message choisi à priori par **A**. Ceci permet à **A** d'avoir une signature de **B** sur un message que **B** n'a jamais vu (d'où le nom de signature aveugle...)
- En fait il s'agit d'une faille basée sur la propriété multiplicative de RSA (page 102) qui a été exploitée pour en faire un nouveau procédé de signature.
- Algorithme: Soit  $S_B$  la signature de RSA de **B** avec  $(n,e)$  et  $d$ , resp. les clés publiques et privées de **B**. Soit  $k$  un entier fixé avec  $\text{pgcd}(n,k) = 1$ :

$$\begin{aligned} f: Z_n &\rightarrow Z_n \text{ avec } f(m) = m \cdot k^e \bmod n & ; \text{blinding function} \\ g: Z_n &\rightarrow Z_n \text{ avec } g(m) = k^{-1} \cdot m \bmod n & ; \text{unblinding function} \end{aligned}$$

ce qui donne:

$$g(S_B(f(m))) = g(S_B(mk^e \bmod n)) = g(m^d k \bmod n) = m^d \bmod n = S_B(m) \quad (*)$$

- Protocole:

$$\begin{aligned} A \rightarrow B: m' &= f(m) \\ A \leftarrow B: s' &= S_B(m') \end{aligned}$$

A calcule  $g(s')$  et obtient la signature souhaitée en utilisant (\*).

## Procédé de Signature d'ElGamal<sup>1</sup>

### Génération des clés

- Chaque entité (**A**) crée une paire de clés (publique et privée) comme suit:
  - **A** génère un nombre premier  $p$  ( $\text{len}(p) \geq 1024$  bits) et un générateur  $\alpha$  de  $Z_p^*$ .
  - **A** génère un nombre aléatoire  $a$ , t.q.  $1 \leq a \leq p-2$  et calcule  $y := \alpha^a \bmod p$ .
  - La clé publique de **A** est  $(p, \alpha, y)$ , la clé privée de **A** est  $a$ .

### Signature

- **A** utilise une fonction de hachage  $H$  et calcule  $m_h := H(m)$ .
- **A** génère un nombre aléatoire  $k$  ( $1 \leq k \leq p-2$  et  $\text{pgcd}(k,p-1) = 1$ ) et calcule  $k^{-1} \bmod (p-1)$
- **A** calcule  $r := \alpha^k \bmod p$  et ensuite  $s := k^{-1} (m_h - ar) \bmod (p-1)$
- La signature de **A** sur le message  $m$  est le couple  $(r,s)$ .

### Vérification

- L'entité **B** obtient  $(p, \alpha, \alpha^a \bmod p)$ , la clé publique authentique de **A**.
- **B** vérifie que  $1 \leq r \leq p-2$ , sinon rejette la signature.
- **B** calcule  $v_1 := y^r r^s \bmod p$ .
- **B** calcule  $H(m)$  et  $v_2 := \alpha^{H(m)} \bmod p$
- **B** accepte la signature ssi.  $v_1 = v_2$ .

- Preuve que le schéma fonctionne: Si  $s \equiv k^{-1} (m_h - ar) \pmod{p-1}$ , on a que:

$$m_h \equiv (ar + ks) \bmod (p-1) \text{ et}$$

$$\mathbf{v}_2 = \alpha^{\mathbf{H}(\mathbf{m})} \mathbf{mod} \mathbf{p}$$

si, comme on souhaite montrer  $\mathbf{m}_h = H(\mathbf{m})$ , en réduisant les exposants  $\text{mod } (p-1)$  (page 92), on peut réécrire  $v_2$ :

$$v_2 \equiv \alpha^{ar+ks} \pmod{p}$$

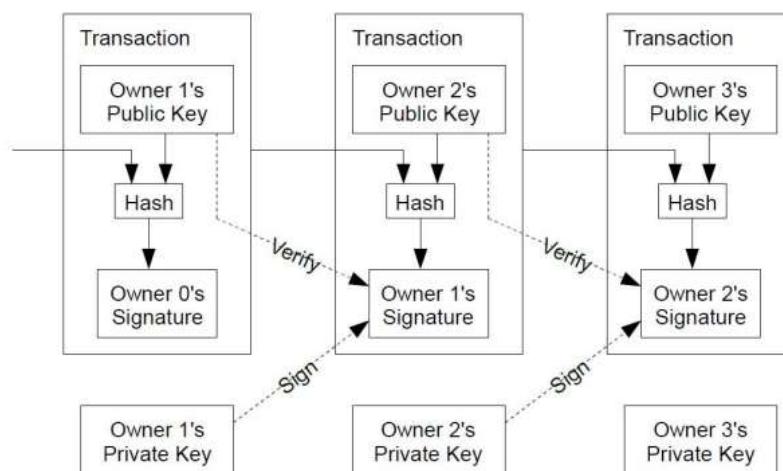
D'autre part:

$$v_1 = y^r r^s \equiv \alpha^{ar} \alpha^{ks} \equiv \alpha^{ar+ks} \pmod{p} \quad \text{c.q.f.d.}$$

- Par construction, le schéma d'ElGamal fonctionne uniquement avec appendice (résultat de l'application d'une fonction de hachage). Le schéma de Nyberg-Rueppel<sup>1</sup> introduit une variation permettant la reconstitution du message.
  - Le *Digital Signature Algorithm (DSA)*, approuvé par le *US National Institute of Standards and Technology* est devenu le standard de signature le plus couramment utilisé. Il est construit sur la base d'un dérivé direct du schéma d'ElGamal avec la fonction de hachage *SHA-1* (page 128).

Signatures Digitales: *Crypto-monnaies*

- La plupart des crypto-monnaies se basent sur la cryptographie asymétrique. Le **bitcoin** p.ex. utilise des signatures digitales pour authentifier ses transactions
  - La dépense ou la transmission de bitcoins nécessite la signature avec la clé privée du détenteur (qui était à son tour le destinataire de la transaction précédente):



*Source Image: Bitcoin: A Peer-to-Peer Electronic Cash System.* Satoshi Nakamoto

- Bitcoin et Ethereum utilisent l'algorithme **ECDSA** (*Elliptic Curve Digital Signature Algorithm*) dérivé de l'algorithme de signature de ElGamal sur les courbes elliptiques dont la sécurité repose sur *ECDLP* (page 109).

### Authentification des données :

- 1) MAC avec une clé symétrique k connue de A et B:

$A \rightarrow B: X, MAC_k(X)$

Si B calcule de son côté  $MAC_k(X)$  et obtient la même valeur => le message provient de A.

- 2) MDC + cryptage symétrique (clé k connue de A et B)

$A \rightarrow B: X, E_k(MDC(X))$

B calcule  $MDC(X)$  et puis  $E_k(MDC(X))$ . Si égal => message vient de A.

- 3) Comme 2) avec confidentialité de X en plus:

$A \rightarrow B: E_k(X, MDC(X))$

- 4) MDC + signature digitale:

$A \rightarrow B: X, Sig_{priv-A}(MDC(X))$

B calcule  $MDC(X)$  et vérifie  $Sig_{priv-A}(MDC(X))$  avec une copie authentique de pub-A. Si égalité => A est à l'origine du message.

Cette solution offre en plus la non-répudiation d'origine.

- Ces protocoles simples n'offrent aucun support sur l'unicité ni sur l'actualité (*timeliness*) des messages reçus et sont exposés à des *replay attacks*! Ils nécessitent des mécanismes tenant compte du temps ou du contexte de la transaction (cf. authentification d'entités).

## Authentification d'Entités: Introduction

- authentification d'entités (*entity authentication*), aussi appelé identification

- Objectifs d'un protocole d'identification robuste:

- 1) Si A et B sont "honnêtes": si A est capable de s'authentifier auprès de B, B doit accepter l'identité de A.
- 2) B ne peut pas réutiliser l'information remise par A pour s'identifier en tant que A auprès de C.
- 3) La probabilité qu'une tierce entité C réussisse à se faire passer par A auprès de B est négligeable.
- 4) Le point 3) reste vrai même si:

- C a observé un grand nombre (polynomial) d'instances du protocole d'identification entre A et B
- C a participé (ev. en se faisant passer par quelqu'un d'autre) à des exécutions précédentes du protocole d'identification auprès de A ou (non exclusif) B.
- plusieurs instances du protocole (ev. initiées par C) peuvent s'exécuter simultanément sans compromettre le processus d'identification.

- Les protocoles d'*authentification faible* satisfont les points 1) et 3). Alors que les protocoles d'*authentification forte* satisfont (au moins partiellement) les points 2) et 4) en plus.

- Terminologie: pour la suite on appelle l'*utilisateur* (*A*), l'entité devant prouver son identité (*claimant*) et le *système* (*B*) l'entité chargée de vérifier cette identité (*verifier*)
- Eléments de base pour l'authentification:
  - *something known*: passwords, PINs, clés privées ou secrètes, etc.
  - *something possessed*: passeport, carte à puces, générateurs de passwords, etc.
  - *something inherent to the human individual*: propriétés *biométriques* comme les empreintes digitales, la rétine, le code ADN, etc.
- **Authentification faible (weak authentication)**: L'utilisateur présente un couple (*userid*, *password*) au système afin de s'identifier. Le *userid* étant l'identité prétendue et le *password* l'évidence corroborant.
- **Authentification forte (strong authentication)**: Contrairement à l'authentification faible, le secret permettant de corroborer l'identité n'est pas révélé explicitement mais, plutôt, l'utilisateur fournit au système une preuve de possession de ce secret.
- Authentification par *zero knowledge*: Ce sont des protocoles d'authentification forte qui ont en plus la caractéristique de prouver l'identité de l'utilisateur sans dévoiler aucune information (ni même une piste...) sur le secret lui-même. En d'autres mots, il s'agit de donner une preuve d'une assertion sans en révéler le moindre détail.

**Relire vite fait p160**

## Authentification Faible

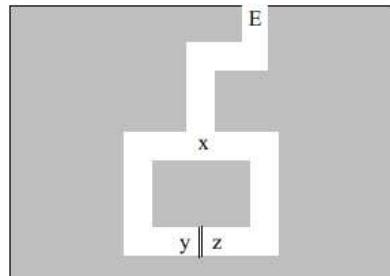
- Les systèmes d'authentification faible sont divisés en deux catégories principales:
  - *Password fixe*: Le password ne dépend pas du temps ni du nombre de fois que le protocole d'identification a été exécuté. Cette catégorie inclue les systèmes où le password est changé par décision de l'utilisateur ou par mesure de sécurité du système.
  - *Password variable*: La modification du password en fonction du temps et/ou du nombre d'exécutions fait partie du protocole d'identification.
- Techniques de stockage propres aux systèmes à password fixe:
  - stockage du password *en clair* dans un fichier protégé par les mécanismes de contrôle d'accès propres au système d'exploitation.  
Problèmes: failles dans le OS, priviléges du “super-user”, *backups*, etc.
  - stockage du password encrypté ou après l'application d'une *one-way function* (éventuellement en rendant publique l'accès à ce fichier, cf. exemple UNIX).  
Problèmes: attaques *off-line*, ie. *guessing attacks*, *brute-force dictionary attacks*, *identification de collisions*, etc.
- Problème le plus grave du *password fixe*: il peut être rejoué après avoir écouté une instance d'identification sur un réseau non protégé !

- Les protocoles d'authentification forte utilisent des techniques cryptographiques symétriques ou asymétriques. On commence par les symétriques...

### **Zero Knowledge Proof :**

- Problème avec les méthodes d'authentification “classiques”: B (ou même un observateur) est en mesure d'obtenir des informations sur le secret détenu par A:
  - Dans les méthodes d'authentification faible (par *password*) c'est le secret dans son intégrité qui est dévoilé.
  - Dans les méthodes *challenge and response* classiques, B peut obtenir des couples [*plaintext / ciphertext*] pouvant servir à la cryptanalyse.
- *Définition:* Un protocole interactif est une **preuve de connaissance** (*proof of knowledge*) lorsqu'il a les deux caractéristiques suivantes:
  - **consistance** (*completeness*): si A et B sont deux entités “honnêtes”, B accepte la preuve fournie par A.
  - **significativité** (*soundness*): Si une entité “malhonnête” C est capable de “tromper” B alors C détient le secret de A (ou une information *polynomialement* équivalente au secret). Ceci équivaut à exiger la possession du secret pour la réussite de la preuve.
- Une preuve de connaissance interactive est dite “*sans apport d'information*” (**zero knowledge interactive proof ou ZKIP**) si elle a, *en plus*, la propriété que A est capable de convaincre B sur un fait sans ne révéler *aucune information* sur le secret qu'elle possède.
- Un protocole est une **ZKIP calculatoire** (*computational ZKIP*) si un observateur capable d'effectuer des tests probabilistes en temps polynomial n'est pas capable de distinguer une preuve authentique (ou A répond) d'une preuve simulée (p.ex. par un générateur aléatoire).
- Un protocole est une **ZKIP parfait** (*perfect ZKIP*) s'il n'existe aucune différence (au sens probabiliste) entre la vraie preuve et la preuve simulée. L'absence d'information dans la preuve est garantie par la *théorie de l'information* de Shannon et non pas par des critères calculatoires.
- Structure générique d'une ZKIP:
  - (1) A → B: témoin (*witness*)
  - (2) A ← B: défi (*challenge*)
  - (3) A → B: réponse (*response*)
  - (1) A choisit un nombre aléatoire secret et envoie à B une preuve de possession de ce secret. Ceci constitue un engagement de la part de A et définit une classe de questions à laquelle A prétend savoir répondre.
  - (2) Le défi envoyé par B choisit (aléatoirement) une question dans cette classe.
  - (3) A répond (en utilisant son secret).
- Si nécessaire, le protocole est répété afin de réduire au maximum la probabilité qu'un “imposteur” devine “par chance” les réponses correctes.

- Cet exemple est décrit dans [Qui89]<sup>1</sup>. Admettons que A connaît un passage entre y et z (le secret).



- (1) B se tient à l'entrée de la caverne au point E.
- (2) A choisit une direction et se dirige vers les points y ou z (choix de témoin).
- (3) Une fois A à l'intérieur de la caverne, B entre à son tour mais s'arrête au point x.
- (4) B demande à A de se rendre au point x par la droite ou par la gauche (le défi).
- (5) En utilisant le secret pour passer de y à z (ou réciproquement) si nécessaire, A obéit aux instructions de B.

Répéter les points 1 à 5 n fois. Si A ne connaît pas le secret, il a une probabilité de  $2^{-n}$  de réussir à tromper B (de deviner “juste”).

- Dans cet exemple, B constate que A peut traverser à volonté le passage yz mais n'obtient aucune information sur la manière de le faire même si le protocole est exécuté des millions de fois.
- Par ailleurs, B ne peut pas convaincre B' du fait que A connaît le secret (comme il aurait été le cas si A encryptait une information en utilisant une clé privée, p.ex.). B' pourrait suspecter A et B d'avoir convenu les séquences (droite/gauche)
- Ce genre de protocoles sont inspirés de la technique du “*cut and choose*” où A et B partagent équitablement une tarte en suivant les étapes suivantes:
  - A coupe la tarte.
  - B choisit un morceau.
  - A prend le morceau restant.
- Le premier ZKIP a été publié en 1985 par S. Goldwasser [Gol85]<sup>1</sup>. L'application du paradigme du *cut and choose* aux protocoles cryptographiques est due à Rabin [Rab78]<sup>2</sup>.

### Key etablisssment protocole :

- Un protocole d'établissement de clés (*key establishment protocol* ou *KEP*) est celui qui met à disposition des entités impliquées un secret partagé (une clé) qui servira comme base pour des échanges cryptographiques ultérieurs.
- Les deux variantes des *KEP* sont les protocoles de transport de clé (*key transport protocol* ou *KTP*) et les protocoles de mise en accord (*key agreement protocol* ou *KAP*).
  - Un *key transport protocol* (*KTP*) est un mécanisme permettant à une entité de créer une clé secrète et de la transférer à son (ses) correspondant(s).
  - Un *key agreement protocol* (*KAP*) est un mécanisme permettant à deux (ou plusieurs) entités de dériver une clé à partir d'informations propres à chaque entité.
- *Key pré-distribution schemes* sont ceux où les clés utilisées sont entièrement déterminées à priori (p.ex. à partir des calculs initiaux).
- *Dynamic key establishment schemes* (*DKE*) sont ceux où les clés changent pour chaque exécution du protocole.

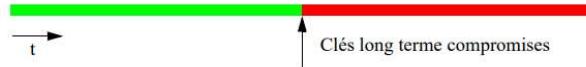
| Key Establishment Protocols |                   |                   |
|-----------------------------|-------------------|-------------------|
| Key Agreement               | Key Transport     |                   |
| symétrique + pré-dist.      | symétrique + DKE  | symétrique + DKE  |
| asymétrique + pré-dist.     | asymétrique + DKE | asymétrique + DKE |

### Propriétés :

- **Implicit key authentication** (ou *key authentication*): propriété par laquelle une entité est assurée que seul(s) son (ses) correspondant(s) peut (peuvent) accéder à une clé secrète. Cependant, ceci ne spécifie rien sur le fait de posséder effectivement la clé.
- **Key confirmation**: propriété permettant à une entité d'être sûre que ses correspondants sont en possession des clés de session générées
- **Explicit key authentication**: = *implicit key authentication + key confirmation*.
- Un **authenticated KEP** est un *KEP* capable de fournir *key authentication*.
- Attaques:
  - Une **attaque passive** est celle qui essaye de démonter un système cryptographique en se limitant à l'enregistrement et à l'analyse des échanges.
  - Une **attaque active** fait intervenir un adversaire qui modifie ou injecte des messages.
  - Un protocole est dit vulnérable à un **known-key attack** si lorsqu'une clé de session antérieure est compromise, il devient possible: (a) de compromettre par une attaque passive des clés futures et/ou (b) de monter des attaques actives visant l'usurpation d'identité.

La plupart des protocoles modernes garantissent les propriétés suivantes:

- **Perfect Forward Secrecy (PFS)** est une caractéristique qui garantit la confidentialité des clés de sessions utilisées par le passé même si les clés long terme (par exemple la clé privée du destinataire) est compromise:



- **Future Secrecy:** Le protocole garantit la sécurité des échanges ultérieurs (les clés des sessions futures sont protégées) même si les clés long terme sont compromises **par un attaquant passif**:



- **Deniability / Repudiability (répudialibilité):** A l'image des protocoles d'authentification Zéro-Knowledge, permet aux entités de garantir l'authentification des échanges sans apporter des informations qui permettraient de prouver à un tiers leur participation dans l'échange cryptographique.

### Diffie-hellman :

- Il permet à deux entités qui ne se sont jamais rencontrées de construire une clé partagée en échangeant des messages sur un canal non confidentiel.
- Protocole:

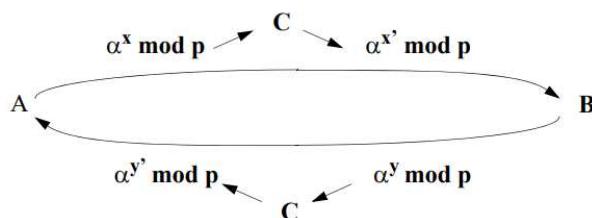
Initialisation: Un nb. premier  $p$  est généré et un générateur  $\alpha$  de  $Z_p^*$ , t.q.  $\alpha \in Z_{p-1}$ . Les deux nombres sont rendus publics.

- (1)  $A \rightarrow B: \alpha^x \text{ mod } p$  ; A choisit un secret  $x \in Z_{p-1}$  et envoie la partie publique
  - (2)  $A \leftarrow B: \alpha^y \text{ mod } p$  ; B choisit un secret  $y \in Z_{p-1}$  et envoie la partie publique
- A calcule la clé secrète:  $K := (\alpha^y)^x \text{ mod } p$  et B à son tour:  $K := (\alpha^x)^y \text{ mod } p$

- La sécurité de ce schéma réside dans l'impossibilité de trouver  $\alpha^{xy} \text{ mod } p$  à partir de  $\alpha^x \text{ mod } p$  et  $\alpha^y \text{ mod } p$ . (*Diffie-Hellman Problem: DHP*).

- **Résultat prouvé:**  $DHP \leq_p DLP$ .

- Diffie-Hellman est sûr (autant que DHP) contre des attaques passives. En d'autres mots, un adversaire qui se limite à voir passer des messages ne peut pas trouver la clé K.
- Ceci n'est cependant plus vrai pour des attaques actives; voyons ce que C peut faire en modifiant les messages:



- C échange des clés secrètes avec A et B, respectivement:  $\alpha^{x'y'} \text{ mod } p$  et  $\alpha^{x'y} \text{ mod } p$  (C contrôle  $x'$  et  $y'$ ). Si C ré-encrypte chaque paquet qu'il reçoit avec la clé publique correspondante, l'attaque se fera de manière transparente pour A et B.
- Cette attaque est appelée *Man in the Middle* (MIM) et s'applique à tous les protocoles asymétriques.
- Elle est due au manque d'authentification des clés publiques, ie. lorsque A "parle" à B, il doit utiliser la clé publique authentique de B.

- Caractéristiques de Diffie-Hellman (non authentifié):
  - *Entity Authentication*: NON.
  - *Implicit key authentication*: NON (par l'attaque *MIM*).
  - *Key confirmation*: NON (dû au risque de *MIM*, A ne peut pas être sûr que B possède la clé secrète partagée).
- Génération de clés symétriques à partir d'une clé partagée Diffie-Hellman:
  - Les quantités manipulés dans DH (notamment K) sont de taille 512 - 1024 bits (suivant le nb. premier p utilisé).
  - Une approche intuitive pour générer des clés symétriques de petite taille (64 - 128 bits) serait de prendre un sous-ensemble de bits de la clé K.
  - Malheureusement, on peut prouver que les clés DH ne sont pas *bit secure* ce qui signifie que des sous ensembles de bits (notamment les *Least Significant Bits*) peuvent être calculés avec un effort non proportionnel à l'effort nécessaire à calculer la clé entière.
  - Pour générer des clés de manière sûre il est conseillé d'appliquer un MDC (comme SHA ou MD5) à *toute* la clé (ev. enchaîner l'application des MDCs pour obtenir des clés symétriques successives).
  - Cette méthode permet d'obtenir un KAP avec *Dynamic Key Establishment*.

## Diffie-Hellman sur des Courbes Elliptiques

### Génération des clés

- A et B choisissent une courbe elliptique  $E_p$  avec  $p$ , un nombre premier de grande taille ( $\text{len}(p) \sim 200$  bits) et un point  $P_0 \in E_p$  de grand ordre (ev. un générateur de  $E_p$ ).
- A génère un nombre aléatoire  $x$ , t.q.  $1 < x < p$  et calcule la partie publique  $xP_0$  (multiplication par un scalaire sur  $E_p$ , pour laquelle, il existe des algorithmes efficaces).
- B génère un nombre aléatoire  $y$  t.q.  $1 < y < p$  et calcule la partie publique  $yP_0$ .
- Protocole d'échange de clés:
  - (1)  $A \rightarrow B: xP_0$
  - (2)  $A \leftarrow B: yP_0$
- Après l'échange A calcule  $K_a := x(yP_0)$  et B calcule à son tour:  $K_b := y(xP_0)$ .
- La propriété commutative des opérations sur  $E_p$  garantit l'égalité:  $K_a = K_b$ .
- La clé secrète partagée résulte d'un processus de sélection de bits de la clé  $K_a$  ou du résultat d'un MDC (fonction de hachage) appliqué à la clé  $K_a$ .
- Il est également nécessaire d'authentifier les parties publiques échangées afin d'éviter les attaques *man-in-the middle* précédemment décrites.
- Les propriétés du protocole sont identiques au cas  $Z_p^*$  (page 192).