

Fonctions de Hachage et Signatures Numériques

DOBA CHRISTIAN NDOUYANG
FOPA NDÉ MILENE LÉA
DJOKAM MPOMO FRANCK CHARES
NDAM MBOMBO RAMINE DELYAN
MBELLE MASSENGUE BOSTIN IVAN
EYENGA ZAMBO ANDY MIGUEL
YOGO KAM CALVIN KARIS

Sous l'encadrement de:
Dr Hervé TALE KALACHI

École Nationale Supérieure Polytechnique de Yaoundé
Département de Génie Informatique

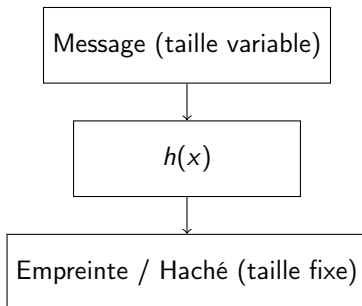


Qu'est-ce qu'une Fonction de Hachage ?

Definition

Une fonction de hachage est un procédé qui transforme une donnée de longueur arbitraire en une valeur de longueur fixe, appelée **empreinte** ou **haché**.

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



- Rapide à calculer.
- Déterministe (le même message donne toujours la même empreinte).

Fonctions de hachage - Motivations

- Transformer un message de longueur arbitraire en une empreinte de taille fixe.
- Détection d'altérations : changement minime du message \Rightarrow empreinte très différente.
- Vérification d'intégrité pour stockage/transmission.
- Accélérer signatures : on signe l'empreinte au lieu du document complet.
- Protection des mots de passe (stockage d'empreintes, salage).
- MACs (hachage + clé) pour authentifier messages.
- Dérivation/diversification de clés, engagements, PRNGs, structures vérifiables (arbres de Merkle).



Propriétés classiques d'une fonction de hachage cryptographique

Résistance à la préimage (One-Way)

À partir d'une empreinte y , il est impossible (meilleure complexité $O(2^n)$) de retrouver le message original x tel que $h(x) = y$.

Résistance à la seconde préimage

Étant donné un message x , il est impossible (meilleure complexité $O(2^n)$) de trouver un autre message x' tel que $h(x) = h(x')$.

Résistance aux collisions

Il est impossible (meilleure complexité $O(2^{n/2})$) de trouver une paire de messages distincts (x, x') qui produisent la même empreinte.



- **Une approche naturelle** : définir une primitive opérant sur un domaine de taille fixe puis étendre ce domaine.
- Fonction de hachage itérative : découper le message et faire un traitement itératif

Extenseur Merkle-Damgård : principe

- construction itérative fondée sur une fonction de compression $f : \{0, 1\}^r \times \{0, 1\}^n \rightarrow \{0, 1\}^n$.
- **Padding standard** (bit '1', zéros, longueur en fin) pour rendre le message multiple de la taille de bloc.
- **Découpage en blocs** M_i puis itération : $VC_0 = IV$; $VC_i = f(VC_{i-1}, M_{i-1})$; $h(M) = VC_t$.
- **Propriété** : sécurité de h hérite de celle de la compression f (mais attention aux attaques structurelles).
- **Faiblesses connues** : extension de longueur, multi-collisions.



Extenseur Merkle-Damgård : Diagramme

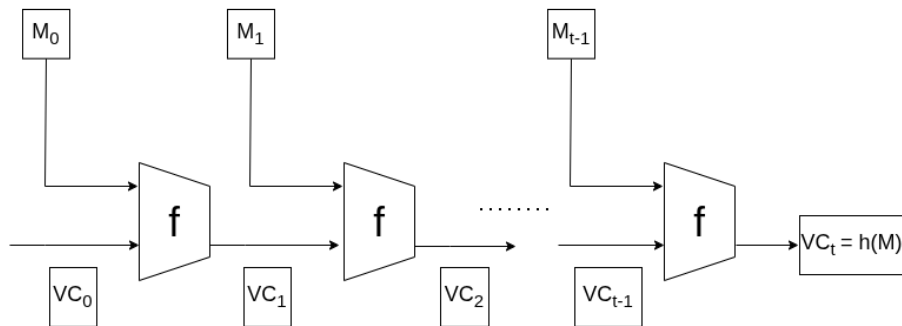


Figure – Diagramme pour la construction Merkle-Damgård

Extenseur « éponge » : principe

- **État interne** de taille $b = r + c$ (r = rate, c = capacity).
- **Fonction de permutation** $p : \{0, 1\}^b \rightarrow \{0, 1\}^b$ (bijective, pseudo-aléatoire).
- Deux phases : **absorption** (XOR du bloc sur la partie rate + application de p) puis **squeezing** (extraction r bits de la partie rate, éventuellement répéter p pour plus de sortie).
- **Avantages** : sortie de longueur variable, résistance à l'extension de longueur si bien paramétré, conception *proche d'un oracle aléatoire*.
- **capacity** $c \rightarrow$ sécurité : typiquement la résistance est liée à $c/2$ pour collisions.
- **rate** $r \rightarrow$ performance : plus $r \rightarrow$ plus rapide mais capacity diminue.
- **Utilisée par SHA-3 / Keccak.**



Extenseur « éponge » : Diagramme

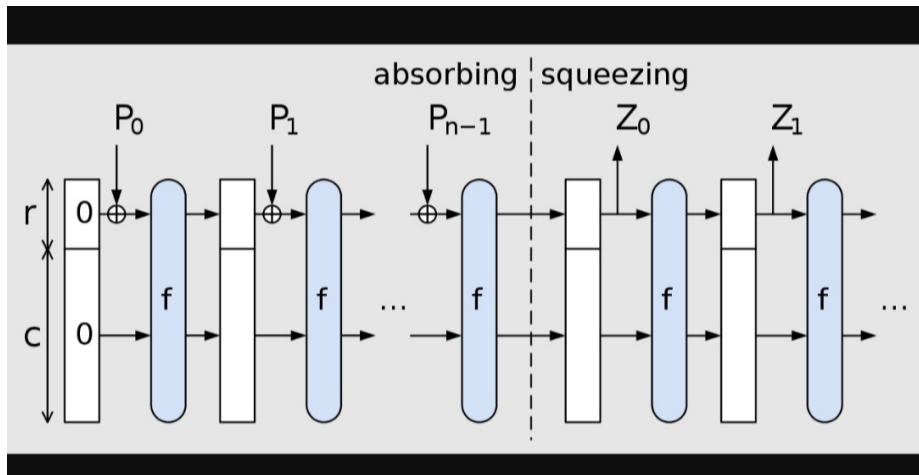


Figure – Diagramme pour l'extenseur éponge

- Condensé de 128 bits ; opérant sur mots de 32 bits ; blocs de 512 bits.
- Suit la construction Merkle-Damgård
- Quatre fonctions logiques non linéaires :

$$F(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z),$$

$$G(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \neg Z),$$

$$H(X, Y, Z) = X \oplus Y \oplus Z,$$

$$I(X, Y, Z) = Y \oplus (X \vee \neg Z).$$

- Table de 64 constantes :

$$T[i] = \lfloor 2^{32} \cdot |\sin(i)| \rfloor, \quad 1 \leq i \leq 64, \text{ i en radian}$$

- Paramètres des rotations cycliques à gauche :

$$s = \begin{cases} \{7, 12, 17, 22\} & \text{(tour 1),} \\ \{5, 9, 14, 20\} & \text{(tour 2),} \\ \{4, 11, 16, 23\} & \text{(tour 3),} \\ \{6, 10, 15, 21\} & \text{(tour 4).} \end{cases}$$



- 4 tampons de travail A, B, C et D de tailles 32 bits

ÉTAPES

- 1 Padding : '1' + k zéros jusqu'à $(L + 1 + k) \equiv 448 \pmod{512}$, puis longueur L codée sur 64 bits (little-endian).
- 2 Initialisation des tampons :

$$A_0 = 67452301,$$

$$B_0 = \text{efcdab89},$$

$$C_0 = 98badcfe,$$

$$D_0 = 10325476.$$

- 3 Pour chaque bloc de 512 bits : extraire 16 mots $X_0..X_{15}$, exécuter 64 itérations (4 tours) :

Tour 1 : $0 \leq j \leq 15$

$$A \leftarrow B + \text{ROTL}_{s_j}(A + F(B, C, D) + X_j + T[j]).$$

Tour 2 : $16 \leq j \leq 31$

$$A \leftarrow B + \text{ROTL}_{s_j}(A + G(B, C, D) + X_{(5j+1) \bmod 16} + T[j]).$$



Tour 3 : $32 \leq j \leq 47$

$$A \leftarrow B + \text{ROTL}_{s_j}(A + H(B, C, D) + X_{(3j+5) \bmod 16} + T[j]).$$

Tour 4 : $48 \leq j \leq 63$

$$A \leftarrow B + \text{ROTL}_{s_j}(A + I(B, C, D) + X_{7j \bmod 16} + T[j]).$$

NB : après chaque opération, les registres sont permutés cycliquement :

$$(A, B, C, D) \leftarrow (D, A, B, C).$$

- À la fin des 64 itérations :

$$A_i = A_{i-1} + A,$$

$$B_i = B_{i-1} + B,$$

$$C_i = C_{i-1} + C,$$

$$D_i = D_{i-1} + D.$$

- Après traitement de tous les blocs** : $h(M) = \text{concaténation } A_n \| B_n \| C_n \| D_n$ (little-endian).
- MD5 cassé pour collisions, vulnérable aux attaques chose-prefix et extension de longueur.



MD5 - Vue d'ensemble

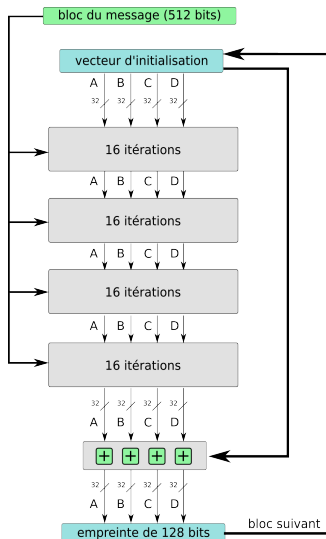


Figure – Vue générale de l'algorithme de MD5, source Wikipédia

MD5 - Une itération

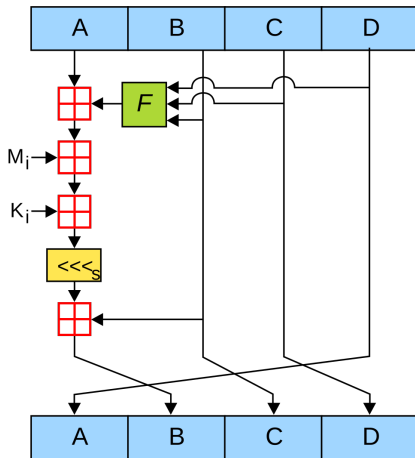


Figure – Vue d'une des 64 itérations de MD5 (X_i et T_i correspondent respectivement à M_i , K_i sur la figure, source Wikipédia)

- Membre de la famille SHA-2; condensé de 256 bits; mots de 32 bits; blocs de 512 bits.
- Suit la construction Merkle-Damgård
- Six (6) fonctions logiques :
 - $CH(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
 - $MAJ(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
 - $BSIG_0(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
 - $BSIG_1(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
 - $SSIG_0(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
 - $SSIG_1(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$



- table de 64 constantes de 32-bits K_0, \dots, K_{63} : premiers 32 bits des parties fractionnaires des racines cubiques des 64 premiers nombres premiers
- 8 tampons de 32 bits représentant la variable de chaînage

ÉTAPES

- 1 Padding : bit '1', K zéros tel que $(L+1+K) \equiv 448 \pmod{512}$, puis longueur L du message sur 64 bits, big-endian.
- 2 Initialisation de la variable de chaînage VC_0 : mots obtenus en prenant les premiers 32 bits des parties fractionnaires des racines carrées des 8 premiers nombres premiers

$$\begin{aligned} VC_0[0] &= 6a09e667, & VC_0[1] &= bb67ae85, \\ VC_0[2] &= 3c6ef372, & VC_0[3] &= a54ff53a, \\ VC_0[4] &= 510e527f, & VC_0[5] &= 9b05688c, \\ VC_0[6] &= 1f83d9ab, & VC_0[7] &= 5be0cd19. \end{aligned}$$



- ③ Traitement itératif des blocs $M_i, 1 \leq i \leq n$: 8 mots de travail a, c, \dots, h ; une suite (W_t) de 64 mots ; VC ; deux tampons T_1, T_2 de 32 bits.

On considère un bloc M_i de 512 bits :

- **Etape 1 : extension de M_i ou construction de (W_t)**

M_i découpé en 16 mots $\rightarrow W_0, \dots, W_{15}$.

$\forall t, 16 \leq t \leq 63$:

$$W_t = (\text{SSIG}_1(W_{t-2}) + W_{t-7} + \text{SSIG}_0(W_{t-15}) + W_{t-16}) \bmod 2^{32}.$$

- **Etape 2 - Initialisation des mots de travail**

$$a = VC_{i-1}[0], \quad b = VC_{i-1}[1],$$

$$c = VC_{i-1}[2], \quad d = VC_{i-1}[3],$$

$$e = VC_{i-1}[4], \quad f = VC_{i-1}[5],$$

$$g = VC_{i-1}[6], \quad h = VC_{i-1}[7].$$



-
- **Etape 3 - calcul du haché intermédiaire VC_i**

Pour $t = 0$ à 63 :

$$T_1 = h + \text{BSIG}_1(e) + \text{CH}(e, f, g) + K_t + W_t$$

$$T_2 = \text{BSIG}_0(a) + \text{MAJ}(a, b, c)$$

$$h \leftarrow g$$

$$g \leftarrow f$$

$$f \leftarrow e$$

$$e \leftarrow d + T_1$$

$$d \leftarrow c$$

$$c \leftarrow b$$

$$b \leftarrow a$$

$$a \leftarrow T_1 + T_2.$$



- **Etape 4 - Mise à jour de VC_i**

$$VC_i[0] = VC_{i-1}[0] + a, \quad VC_i[1] = VC_{i-1}[1] + b$$

$$VC_i[2] = VC_{i-1}[2] + c, \quad VC_i[3] = VC_{i-1}[3] + d$$

$$VC_i[4] = VC_{i-1}[4] + e, \quad VC_i[5] = VC_{i-1}[5] + f$$

$$VC_i[6] = VC_{i-1}[6] + g, \quad VC_i[7] = VC_{i-1}[7] + h.$$

- **Après traitement de tous les blocs :**

$$h(M) = VC_n[0] \parallel VC_n[1] \parallel \dots \parallel VC_n[7]$$

SHA-256 - Diagramme de vue générale

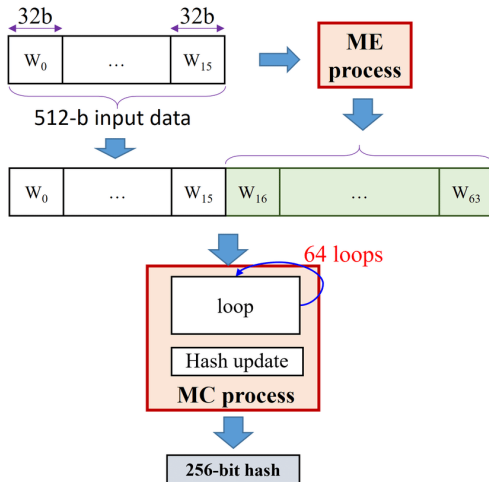


Figure – Vue générale de l'algorithme SHA-256

SHA-256 : une itération

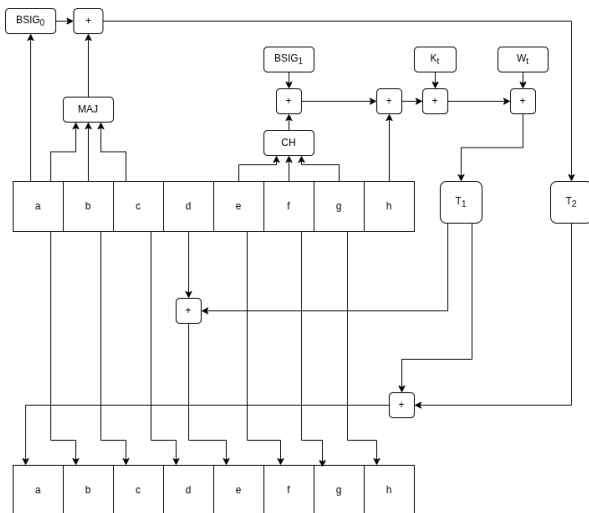


Figure – Vue d'une itération du SHA-256

Definition

C'est un code cryptographique qu'un émetteur attache à un message numérique et permettant d'authentifier l'origine et garantir l'intégrité du message.

Motivations ?

- Authentification de l'émetteur
- Intégrité du message
- Non-répudiation
- Vérification indépendante et publique : environnements distribués, ouverts, transparents (Internet, blockchain)
- Preuve d'engagement ou d'accord : transaction financière, système de vote électronique, etc



Le Moteur des Signatures : Cryptographie Asymétrique et PKI

La Paire de Clés

(Clé Privée, Clé Publique)

Clé privée → Gardée secrète.

Clé publique → partagée librement

Avantages :

- Non répudiation
- confidentialité plus difficile à compromettre
- Une seule paire mais multiples canaux bidirectionnels indép.

La Gestion de la Confiance

Comment savoir si une clé publique appartient bien à une personne ?

Solution : PKI (Infrastructure à

Clés Publiques)

- Autorité de Certification (AC)
- Certificat Numérique



Le Moteur des Signatures : Cryptographie Asymétrique et PKI

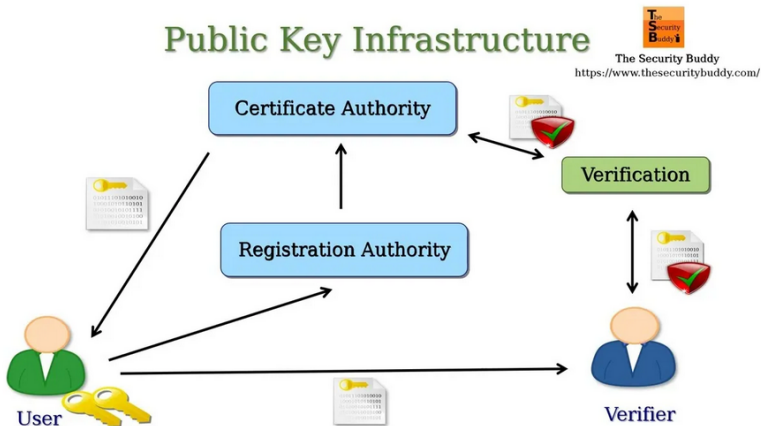


Figure – Illustration PKI

Algorithme de Signature n°1 : RSA

La sécurité repose sur la difficulté de factoriser de très grands nombres premiers ($n = p * q$).

Génération de clés

- Choisir deux grands premiers secrets p et q .
- Calculer $n = pq$ et $\varphi(n) = (p - 1)(q - 1)$.
- Choisir e tel que $1 < e < \varphi(n)$ et $\gcd(e, \varphi(n)) = 1$ (valeur courante : $e = 65537$).
- Calculer d tel que $ed \equiv 1 \pmod{\varphi(n)}$ (par exemple via l'algorithme d'Euclide étendu).
- La clé publique est (n, e) ; la clé privée est d (et, pour des optimisations, les facteurs p, q sont conservés).



Cycle de vie de signature

1 Hachage

La fonction de hachage h est publique.

Le message M est d'abord haché pour obtenir $h(M)$.

2 Signature (avec la clé privée d)

$$\text{Signature} = (h(M))^d \mod n$$

3 Message envoyé : (M, Signature)

4 Vérification (avec la clé publique e)

$$h(M)' = (\text{Signature})^e \mod n$$

5 Validation

Si $h(M)' = h(M)$, la signature est **Valide** ✓.



Algorithme de Signature n°2 : DSA (Digital Signature Algorithm)

La sécurité du DSA repose sur la difficulté du problème du logarithme discret dans un groupe fini.

Trois étapes pour signer :

- ➊ Génération des clés
- ➋ Signature du document
- ➌ Vérification du document signé



Génération de clés

- 1 Choisir des longueurs L et N avec L divisible par 64. Ces longueurs définissent directement le niveau de sécurité de la clé. NIST 800-57 recommande de choisir $L = 3072$ et $N = 256$ pour une sécurité équivalente à 128 bits.
- 2 Choisir un nombre premier p de longueur L
- 3 Choisir un nombre premier q de longueur N , de telle façon que $p - 1 = qz$ avec z un entier
- 4 Choisir h , avec $1 < h < p - 1$ de manière que :

$$g = h^z \bmod p > 1$$

- 5 Générer aléatoirement un x , avec $0 < x < q$
- 6 Calculer :

$$y = g^x \bmod p$$

Clés résultantes

- Clé publique : (p, q, g, y)
- Clé privée : x



Signature

1 Choisir un nombre aléatoire s tel que $1 < s < q$

2 Calculer :

$$s_1 = (g^s \bmod p) \bmod q$$

3 Si $s_1 = 0$, recommencer avec un autre s

4 Calculer :

$$s_2 = (H(m) + s_1 \cdot x) \cdot s^{-1} \bmod q$$

où $H(m)$ est le résultat d'un hachage cryptographique

5 Si $s_2 = 0$, recommencer avec un autre s

6 La signature est (s_1, s_2)



Vérification

❶ Rejeter la signature si $0 < s_1 < q$ ou $0 < s_2 < q$ n'est pas vérifiée

❷ Calculer :

$$w = s_2^{-1} \bmod q$$

❸ Calculer :

$$u_1 = H(m) \cdot w \bmod q$$

❹ Calculer :

$$u_2 = s_1 \cdot w \bmod q$$

❺ Calculer :

$$v = (g^{u_1} \cdot y^{u_2} \bmod p) \bmod q$$

❻ La signature est valide si $v = s_1$

Synthèse : de l'empreinte à l'engagement

- **Fonctions de Hachage** : Ce sont les empreintes numériques du monde digital, dont la sécurité repose sur la résistance à la préimage et aux collisions.
- **Constructions et Algorithmes** : Les constructions évoluent (Merkle-Damgård → Éponge) pour contrer les menaces. Le choix de l'algorithme (SHA-256 plutôt que MD5) est critique.
- **Signatures Numériques** : Elles combinent le hachage et la cryptographie asymétrique pour garantir l'authenticité, l'intégrité et la non-répudiation.
- **Conclusion** : La maîtrise de ces briques cryptographiques est essentielle pour concevoir les systèmes fiables et sécurisés de demain.



- Objectif : développer un logiciel illustrant l'utilité des fonctions de hachage et signatures numériques
- Fonctionnalité développée : signer des documents, avec chiffrement.
-

Flux fonctionnel

- 1 L'utilisateur A (Alice) charge un document M ;
- 2 L'application calcule le haché $h(M)$ (SHA-256) ;
- 3 A signe le haché avec sa clé privée (RSA-PSS + SHA-256) pour obtenir la signature S_M ;
- 4 Optionnellement, A chiffre le document/signature (AES-256-GCM / RSA-OAEP) ;
- 5 Le destinataire B (Bob) vérifie le certificat X.509 de A et la signature ;
- 6 Si la vérification est correcte, le document est accepté comme intègre et authentique.



Choix techniques

- Hachage : SHA-256 (performance, standardisation) ;
- Signature asymétrique : RSA-PSS avec SHA-256 pour la robustesse probabiliste ;
- Chiffrement asymétrique : RSA-OAEP pour l'échange de clés ou petites charges ;
- Chiffrement symétrique : AES-256-GCM pour la confidentialité et l'intégrité (auth tag) ;
- Certificats : X.509 v3 pour références d'identité standard ;
- Langage/back-end : Python + Django (rapidité de prototypage, bibliothèques crypto) ;
- Déploiement : plateforme Railway (application publique de démonstration) :



Démonstration

Lien de l'app : <https://projet-si-production.up.railway.app/>



Soit h une fonction de hachage cryptographique produisant une empreinte de 256 bits.

- ❶ Expliquez la différence entre la **résistance à la préimage** et la **résistance aux collisions**.
- ❷ Quelle est la complexité théorique attendue pour trouver :
 - une préimage ?
 - une collision ?
- ❸ Pourquoi la résistance aux collisions est-elle plus difficile à atteindre que la résistance à la préimage sur le plan théorique ?

- ❶ **Résistance à la préimage** : étant donné une empreinte y , il doit être impossible de trouver un message x tel que $h(x) = y$.
Résistance aux collisions : il doit être impossible de trouver deux messages distincts x_1 et x_2 tels que $h(x_1) = h(x_2)$.
- ❷ Complexité théorique (pour une fonction idéale) :
 - Préimage : $O(2^{256})$
 - Collision : $O(2^{128})$ (grâce à l'attaque des anniversaires)
- ❸ La recherche de collision bénéficie de l'**effet du paradoxe des anniversaires** : on cherche une paire parmi N messages, ce qui réduit la complexité à \sqrt{N} , alors que pour une préimage, on cherche un message spécifique correspondant à une empreinte donnée, d'où une complexité en N .

Fonctions de Hachage et Signatures Numériques

FIN

