Table of contents

Fonctions de Hachage Cryptographiques et MACs

Fonctions à Sens Unique (One-Way Functions)

Une fonction f est à sens unique si $f(x) = y$ est facile à calculer, mais trouver x à partir de y est calculatoirement impossible pour la majorité des valeurs.

Exemples :

- Carrés modulo composite : $f(x) = x^2 \bmod n$ avec $n = pq$
- Construction DES : $y = E_k(x) \oplus x$ avec k fixée et connue

Note : OWF OWHF (les hash functions imposent compression et 2nd-preimage resistance).

> Texte original
>
> Une **fonction f est dite à sens unique** (one-way function ou **OWF**) si $x \in X$ on peut facilement calculer $f(x) = y$ mais pour la grande majorité des $y \in Y$ il est **calculatoirement impossible** de trouver un x tel que $f(x) = y$.
> **Exemples:**
>
> - calcul des carrées modulo un composite: $f(x) = x^2 \bmod n$ avec $n = pq$ (p et q

inconnus) est une **one-way function** car l'inverse est difficile (voir le problème de base **SQROOTP**).
- on peut construire une one-way function sur la base de DES ou de n'importe quel autre système de cryptage à blocs E comme suit: $y = f(x) = E_k(x) \oplus x$, $\forall x$, avec k une clé fixée et connue. On peut considérer que $E_k(x) \oplus x$ a un comportement (pseudo) aléatoire par construction de E. Le calcul de l'inverse revient à trouver un x tel que: $x = E_k^{-1}(x \oplus y)$, ce qui est considéré difficile avec les propriétés de E. A noter que $f(x) = E_k(x)$ ne suffirait pas pour en faire une OWF car, en connaissant la clé, DES est réversible.

OWF $\neq$ **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.

Révision rapide

OWF : facile dans un sens ($f(x) \to y$), impossible dans l'autre ($y \to x$).
Exemples : carrés modulaires, $E_k(x) \oplus x$.
OWF $\neq$ OWHF (hash functions = plus de contraintes).

Hash Functions : Définitions

Une hash function h possède deux propriétés essentielles :

- **Compression** : transforme des données de longueur arbitraire en sortie de longueur fixe
- **Facilité de calcul** : $h(x)$ est rapide à calculer

Classification :

- **Unkeyed** (sans clé) : MDC (Manipulation Detection Code)
- **Keyed** (avec clé) : MAC (Message Authentication Code)

Propriétés de sécurité :

1. **Preimage resistance** : étant donné y, impossible de trouver x tel que $h(x) = y$
2. **2nd-preimage resistance** (weak collision) : étant donné x, impossible de trouver $x' \neq x$ tel que $h(x) = h(x')$
3. **Collision resistance** (strong collision) : impossible de trouver $x \neq x'$ quelconques avec $h(x) = h(x')$

Terminologie :

- **OWHF** (weak one-way) : satisfait (1) et (2)
- **CRHF** (strong one-way) : satisfait (2) et (3)

Texte original

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 fonction ($h_k(x) = y$); 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) 2nd-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) 2)). Aussi appelée: **strong one-way hash function**.

OWF 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.

Révision rapide

Hash function : compression + calcul facile
MDC (sans clé) pour intégrité
MAC (avec clé) pour authentification
Propriétés

1. preimage resistance
2. 2nd-preimage resistance
3. collision resistance

OWHF = (1)+(2)
CRHF = (2)+(3).

Message Authentication Codes (MACs)

Un MAC est une famille de fonctions h_k paramétrées par une clé secrète k :

Propriétés :

1. **Compression** : entrée arbitraire → sortie fixe
2. **Facile à calculer** : avec k connue, $h_k(x)$ est rapide
3. **Computation-resistance** : sans k, impossible de calculer des paires $(x, h_k(x))$ valides

Implications :

- Key non-recovery : impossible de retrouver k à partir de paires $(x_i, h_k(x_i))$
- Preimage et collision resistance pour quiconque ne possède pas k

Usage : Authentification d'origine + intégrité des messages sans révéler de secret directement.

Texte original

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.

Révision rapide

MAC = hash avec clé k
Sans k : impossible de forger $(x, h_k(x))$ ou retrouver k
Garantit authentification d'origine + intégrité.

Attaques sur les MDCs

Attaque 2nd-Preimage Resistance

Problème : Étant donné $h(x) = y$, trouver x' tel que $h(x') = h(x)$.

Analyse probabiliste :

Pour un digest de m bits ($n = 2^m$ sorties possibles), la probabilité d'avoir au moins une collision après k essais est :

$$P(\text{collision}) \approx 1 - (1 - 1/n)^k \approx k/n$$

Pour $P = 0.5$: $k = n/2 = 2^{m-1}$

Conclusion : Pour un digest de m bits, il faut environ 2^{m-1} essais pour trouver une 2nd-preimage avec probabilité 0.5.

Texte original

Problème: étant donné $h(x) = y$, trouver x' tel que $h(x') = h(x)$.
Exemple pratique: on a un texte avec un digest associé portant une signature digitale; on veut créer un faux texte portant la même signature (sans avoir le contrôle sur le texte original). Quelles sont nos chances d'un point de vue probabiliste?
Soit une hash function h avec n sorties possibles et une valeur donnée $h(x)$. Si h est appliquée à k valeurs aléatoires, quelle doit être la valeur de k pour que la probabilité d'avoir au moins un y tel que $h(x) = h(y)$ soit 0.5?
Pour la première valeur de y, la probabilité que $h(x) = h(y)$ est $1/n$. Inversement, la probabilité que $h(x) \neq h(y)$ est $1 - 1/n$. Pour k valeurs, la probabilité de n'avoir aucune collision est de: $(1 - 1/n)^k$, soit:

$$\left(1 - \frac{1}{n}\right)^k = 1 - \frac{k}{n} + \frac{1}{2!}\left(\frac{k}{n}\right)^2 - \frac{1}{3!}\left(\frac{k}{n}\right)^3 + ...$$

ce qui pour n très grand peut être approché par $1 - k/n$. Par conséquent, la probabilité complémentaire d'avoir au moins une collision est d'environ k/n; c'est qui nous donne $k = n/2$ pour une probabilité de 0.5.
Conclusion: pour un digest de m bits, le nombre d'essais nécessaires à trouver un y tel que $h(x) = h(y)$ avec une probabilité de 0.5 est 2^{m-1}.

Révision rapide

Pour casser 2nd-preimage resistance avec digest de m bits : 2^{m-1} essais (prob 0.5).

Attaque Collision Resistance : Birthday Paradox

Problème : Trouver deux valeurs distinctes x, x' telles que $h(x) = h(x')$.

Paradoxe d'anniversaire : Dans un groupe de 23 personnes, probabilité > 0.5 d'avoir deux anniversaires identiques.

Résultat mathématique :

Pour n sorties possibles, la probabilité de collision après k calculs :

$$P(\text{au moins 1 collision}) = 1 - e^{-k(k-1)/(2n)}$$

Pour $P \geq 0.5$: $k \approx 1.17\sqrt{n}$

Conséquence cryptographique : Pour un digest de m bits ($n = 2^m$ sorties), il faut environ $2^{m/2}$ calculs pour trouver une collision avec probabilité > 0.5.

Exemple pratique : Modification d'un contrat en 237 variations pour trouver une version frauduleuse ayant le même digest que la version légitime.

Texte original

Problème: trouver deux valeurs x, x' distincts tel que $h(x) = h(x')$.
Exemple pratique: On doit faire signer un texte à quelqu'un et on veut appliquer cette signature à un texte falsifié (on contrôle le texte original). Quelles sont nos chances de trouver deux textes originaux satisfaisant ce critère?
Le **birthday paradox** est un problème probabiliste classique qui montre que dans une réunion de 23 personnes seulement, on a déjà une chance sur deux d'avoir deux personnes ayant leur anniversaire le même jour.
Soit $y_1, y_2, ..., y_n$ toutes les sorties possibles d'une hash function. Combien des $h(x_i)$: $h(x_1), h(x_2), ..., h(x_k)$ devons nous calculer pour avoir une probabilité de collision égale ou supérieure à 0.5 ?
Le premier choix pour $h(x_1)$ est arbitraire (prob = 1), le deuxième $h(x_2) \neq h(x_1)$ a une probabilité de $1 - 1/n$, le troisième de $1 - 2/n$, etc. Ce qui nous donne une probabilité de ne pas avoir des collisions égale à:

$$P_{\text{no collision}} = \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right)$$

On prouve facilement (développement en série de e^{-x}) que pour $0 \leq x \leq 1$: $1 - x \leq e^{-x}$ et donc:

$$P_{\text{no coll}} \leq \prod_{i=1}^{k-1} \left(1 - \frac{i}{n}\right) \leq \prod_{i=1}^{k-1} e^{-i/n} = e^{-k(k-1)/(2n)}$$

La probabilité d'avoir au moins une collision est $P_{\text{au-moins1}} = 1 - P_{\text{no-coll}}$. Pour connaître la valeur de k pour laquelle $P_{\text{au-moins1}}$ est plus grand que 0.5, il suffit de calculer:

$$\frac{1}{2} \leq 1 - e^{-k(k-1)/(2n)}$$

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

$$k \geq \sqrt{2\ln(2) \cdot n} \approx 1.17\sqrt{n}$$

En prenant $n = 365$ pour l'anniversaire, on obtient $k = 22.3$, ce qui confirme l'énoncé du problème.
Conséquence pour les hash functions: Soit une hash function avec 2^m sorties possibles. Si h est appliqué à $k = 2^{m/2}$ entrées on a une probabilité supérieur à 0.5 d'obtenir $h(x_i) = h(x_j)$.

Révision rapide

Birthday paradox : pour casser collision resistance avec digest de m bits : $2^{m/2}$ essais (prob > 0.5).
Exemple : 23 personnes suffisent pour anniversaires identiques.

Résistance Calculatoire : Récapitulatif

Pour une hash function avec digest de n bits et clé MAC de t bits :

Type	Propriété	Difficulté	Taille conseillée
OWHF	Preimage	2^n	$n \geq 128$ bits
	2nd-preimage	2^{n-1}	
CRHF	Collision	$2^{n/2}$	$n \geq 256$ bits
MAC	Key recovery	2^t	$t \geq 256$ bits
	Computation	$\min(2^t, 2^n)$	$n \geq 128$ bits

Implications pratiques :

- Pour intégrité seule (OWHF) : 128 bits suffisent
- Pour résistance aux collisions (CRHF) : minimum 256 bits
- MACs : clé de 256 bits, digest de 128 bits minimum

Texte original

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

Type de Hash Fct.	Caractéristique	Difficulté Calculatoire	But de l'attaque	Taille conseillée du digest/clé
OWHF	preimage resistance	2^n	trouver une préimage	$n \geq 128$ bits
	2nd-preimage résistance	2^{n-1}	trouver x' avec $h(x') = h(x)$	
CRHF	collision resistance	$2^{n/2}$	trouver une collision	$n \geq 256$ bits
MAC	key non-recovery	2^t	trouver la clé	$n \geq 128$

computation resistance	$\min(2^t, 2^n)$	produire un $(x, h_k(x))$	$t \geq 256$

Révision rapide

Efforts : preimage 2^n, 2nd-preimage 2^{n-1}, collision $2^{n/2}$.
Tailles : OWHF 128 bits, CRHF 256 bits, MAC clé 256 bits.

MDCs Basés sur des Systèmes de Cryptage

Principe : Utiliser un algorithme de cryptage symétrique (DES, AES) pour construire un MDC.

Défis à résoudre :

- Casser la réversibilité des algorithmes symétriques
- Augmenter la largeur nominale (DES = 64 bits insuffisant pour CRHF)

Fonctionnement :

- Traitement séquentiel des blocs
- Opérations de chaînage avec XOR
- Combinaison de n boîtes pour digests de taille $n\times$ largeur nominale

Modèles classiques :

1. **Matyas-Meyer-Oseas** : $H_i = E_{m_i}(H_{i-1}) \oplus H_{i-1}$
2. **Davies-Meyer** : $H_i = E_{m_i}(H_{i-1}) \oplus m_i$
3. **Miyaguchi-Preneel** : $H_i = E_{m_i}(H_{i-1}) \oplus H_{i-1} \oplus m_i$

Exemples pratiques :

- **MDC-2** : utilise 2 boîtes DES → digest 128 bits
- **MDC-4** : utilise 4 boîtes DES → digest 128 bits

Limitation : Sécurité fortement dépendante de l'algorithme sous-jacent.

Texte original

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.
Exemples:

- Les modèles de **Matyas-Meyer-Oseas**, **Davies-Meyer** et **Miyaguchi-Preneel**.
- **MDC-2** et **MDC-4** utilisant respectivement 2 et 4 boîtes DES. Digest = 128 bits.

Révision rapide

MDCs à partir de crypto symétrique : casser réversibilité + chaînage XOR.
Modèles : Matyas-Meyer-Oseas, Davies-Meyer, Miyaguchi-Preneel.
MDC-2/4 avec DES → 128 bits.

Customized MDCs

Fonctions conçues spécifiquement pour la génération de digests, optimisées pour vitesse et sécurité.

Éléments de construction :

- Padding + ajout de la longueur du message
- Constantes prédéfinies pour augmenter la dispersion
- Rounds successifs avec opérations logiques et rotations
- Chaînage des sorties entre rounds
- Chaque bit du digest dépend de chaque bit d'entrée

Algorithmes principaux :

Algorithme	Année	Digest	Statut
MD5	1992	128 bits	Cassé
SHA-0	1993	160 bits	Collisions en 2^{39}
SHA-1	1995	160 bits	Collisions en 2^{63}
SHA-2	-	224-512 bits	Sûr actuellement
SHA-3 (Keccak)	2012	224-512 bits	Standard actuel

Évolution des attaques :

- 2004 : Collisions complètes sur MD5 (X. Wang)
- 2005 : SHA-1 cassé théoriquement (2^{63} opérations)
- 2008 : Création de certificats CA frauduleux via MD5
- 2012 : SHA-3 adopté comme nouveau standard

Texte original

Il s'agit de fonctions conçues exclusivement pour générer des codes d'intégrité (des digests) avec un soucis 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 a 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)

Derniers Développements:

- X.Wang et al. culminent en 2004 un long travail visant à trouver des collisions dans l'algorithme MD5. Ils publient deux paires de collisions pour des messages de 1024 bits.

- En 2005, X.Wang et al. prouvent dans la conférence CRYPTO'05 que le nombre d'opérations nécessaires pour trouver des collisions sur SHA-1 (standard actuel pour les fonctions de hashage sécurisées) est seulement de 2^{63}.
- Ces attaques ont pour cible la recherche de collisions arbitraires mais lors de CRYPTO'06 des chercheurs de l'Université de Graz en Autriche proposent une méthode pour contrôler partiellement le contenu des collisions.
- En Décembre 2008 on montre qu'on peut générer des collisions contrôlées sur MD5 et créer ainsi une Certification Authority illicite permettant des forger des certificats acceptés par n'importe quel browser.
- Ces résultats s'appuient sur des approches **analytiques** (par opposition au brute force!)
- Le processus de sélection de successeur de SHA-1 est semblable à celui ayant désigné AES comme standard de cryptage en blocs. Le NIST a décidé (Octobre 2012) que **Keccak** serait l'algorithme de base pour **SHA-3**.

Révision rapide

Customized MDCs

- MD5 (cassé)
- SHA-0 (cassé)
- SHA-1 (faible)
- SHA-2 (sûr)
- SHA-3/Keccak (standard actuel).

Construction : padding + constantes + rounds + chaînage.

MACs Basés sur des Systèmes de Cryptage

CBC-MAC : Utilise un algorithme de chiffrement par blocs en mode CBC.

Fonctionnement :

- Mode CBC avec IV = 0
- Élimination des ciphertexts intermédiaires
- Seul le dernier bloc chiffré est conservé comme MAC

Avec DES :

- Longueur clé : 56 bits (112 en Triple-DES optionnel)
- Longueur MAC : 64 bits

Avantages :

- Réutilisation de l'infrastructure de chiffrement existante
- Performances acceptables

Limitations :

- Sécurité limitée par la taille du bloc (64 bits pour DES)
- Vulnérable si utilisé incorrectement (ex: sans IV variable)

Texte original

Algorithme CBC-MAC basé sur DES-CBC avec IV = 0 et élimination des ciphertext intermédiaires

- longueur de clé = 56 bits (112 en cas d'utilisation de la partie optionnelle)
- Longueur du MAC-value = 64 bits

Le schéma montre le traitement séquentiel des blocs de message M_1, M_2, M_3 avec l'algorithme de cryptage E et la clé k. Les ciphertexts intermédiaires C_1, C_2 sont éliminés. Seul le dernier bloc C_3 constitue le MAC.

Révision rapide

CBC-MAC : mode CBC + IV=0, seul dernier bloc gardé. DES : clé 56/112 bits, MAC 64 bits.

Nested MACs et HMACs

Nested MAC (NMAC) : Composition de deux familles de MACs G et H :

$$\text{NMAC}_{k,l}(x) = g_k(h_l(x))$$

Sécurité : Dépend de deux critères :

- G résistante aux collisions
- H résistante aux attaques spécifiques MACs

HMAC (standard FIPS 198, 2002) : Nested MAC utilisant des MDCs sans clé (SHA-1, SHA-256).

Construction :

- Constantes : `ipad = 0x363636...36` et `opad = 0x5C5C5C...5C` (512 bits)
- Clé k de 512 bits

$$\text{HMAC-256}_k(x) = \text{SHA-256}((k \oplus \text{opad}) \parallel \text{SHA-256}((k \oplus \text{ipad}) \parallel x))$$

Avantages :

- MACs les plus utilisés en pratique
- Attaques sur SHA plus difficiles avec clé secrète
- Performance excellente
- Standardisé et largement supporté

Texte original

Un **Nested MAC** ou **NMAC** est une composition de 2 familles de fonctions MACs G et H paramètrès 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 dénommés `ipad` et `opad` telles que:

- `opad := 363636 ... 36`
- `ipad := 5C5C5C ... 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.

Révision rapide

HMAC : double hash avec clés dérivées (`ipad`/`opad`). $\text{HMAC}_k(x) = H((k \oplus \text{opad}) \parallel H((k \oplus \text{ipad}) \parallel x))$. Standard, sûr, performant.

Applications des Hash Functions

Intégrité des Données

Trois approches principales :

1. **MAC seul :**
   - $A \to B : X, \text{MAC}_k(X)$
   - Authentification + intégrité garanties
   - Nécessite clé partagée
2. **MDC + Encryption :**
   - $A \to B : E_k(X, \text{MDC}(X))$
   - Confidentialité + intégrité
   - Clé symétrique partagée
3. **MDC + Canal authentique :**
   - $A \to B : X$ (canal normal)
   - $A \to B : \text{MDC}(X)$ (canal authentique)
   - Séparation des canaux

Limitations : Ces protocoles simples n'offrent pas de protection contre les replay attacks.

Solution : ajouter timestamps ou numéros de séquence.

Texte original

MAC Seul:

$$A \to B : X, \text{MAC}_k(X)$$

Si B calcule de son coté $\text{MAC}_k(X)$ et obtient la même valeur le message provient de A.
MDC + cryptage symétrique (clé k connue de A et B)

$$A \to B : X, E_k(\text{MDC}(X))$$

B calcule $\text{MDC}(X)$ et puis $E_k(\text{MDC}(X))$. Si égal message vient de A.
Comme 2) avec confidentialité de X en plus:

$$A \to B : E_k(X, \text{MDC}(X))$$

MDC + signature digitale:

$$A \to B : X, \text{Sig}_{\text{priv-A}}(\text{MDC}(X))$$

B calcule $\text{MDC}(X)$ et vérifie $\text{Sig}_{\text{priv-A}}(\text{MDC}(X))$ 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).

Révision rapide

Intégrité : MAC seul, MDC+crypto, MDC+signature.
Vulnérable aux replay sans timestamps/nonces.

Blockchains et Proof of Work

Bitcoin et blockchains : Utilisation de hash functions pour chaîner les blocs de transactions.

Caractéristiques :

- Transactions publiques et visibles
- Blocs chaînés via fonctions de hachage cryptographiques
- Minage = résolution d'un puzzle cryptographique (proof of work)

Proof of Work :

- Trouver un nonce tel que hash(bloc $\|$ nonce) $<$ target
- Puzzle coûteux en calcul, validation rapide
- Premier mineur à résoudre reçoit récompense en bitcoins

Sécurité :

- Blockchain = registre public, décentralisé, immuable
- Falsification nécessiterait effort $>$ tous mineurs honnêtes
- Protection basée sur propriétés CRHF

Statistiques Bitcoin (octobre 2025) :

- Difficulty : 150.84 T
- Target : $\approx 2^{177}$ (pseudo-collision sur 79 bits)
- Hashrate : ~1.1 ZH/sec (1.1×10^{21} hash/sec)
- Temps moyen génération bloc : 10 minutes

Texte original

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 chaine de blocs résultante (**blockchain**) devient alors un registre publique (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.
Statistiques Bitcoin 13/10/2025:

- **Difficulty**: 150.84 T
- **Target**: 2^{224}/Difficulty $=\approx 2^{177}$. Le digest valable pour générer un bloc doit être inférieur à 2^{177}, ce qui signifie une pseudo-collision sur les 79 bits de poids plus fort. La variation sur les inputs dépend du **nonce**.
- **Hashrate**: ~ 1.1 ZH/sec (1.1×10^{21} hashes /sec)
- **Fonctions de hachage exécutées pour obtenir un bloc**: $\sim 660 \times 10^{21}$
- **Temps moyen de génération d'un bloc**: 10 min

Révision rapide

Blockchain : chaînage de blocs via hash.
Proof of Work : trouver nonce pour hash $<$ target.
Sécurité = effort $>$ tous mineurs.
Bitcoin : ~10 min/bloc, 10^{21} hash/sec.

Autres Applications

1. **Authentification :**
   - Data origin authentication (DOA)
   - Transaction authentication (DOA + paramètres temporels)
2. **Virus checking :**
   - Créateur publie digest = h(logiciel) via canal sûr
   - Utilisateurs vérifient intégrité en recalculant le digest
3. **Distribution des clés publiques :**

- Publier h(clé publique) au lieu de la clé complète
- Vérification d'authenticité simplifiée

4. **Timestamp sur documents :**

- Timestamp appliqué au digest plutôt qu'au document complet
- Réduction des données à signer

5. **One-time password (S-Key) :**

- Chaîne de hash : $x_1 = h(x_0), x_2 = h(x_1), \ldots, x_n = h(x_{n-1})$
- Système stocke x_n, utilisateur fournit x_{n-1}
- Vérification : $h(x_{n-1}) = x_n$
- Après validation, système stocke x_{n-1} pour prochaine fois

Texte original

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'utilsateur rentre x_{n-1}. Si $h(x_{n-1}) == x_n$ OK.
- Le système mémorise alors x_{n-1} et ainsi de suite.

Révision rapide

Applications

- authentification
- virus checking
- distribution clés publiques
- timestamp
- one-time passwords (chaîne de hash)

Randomized Hash Functions : Exemple UNIX

Problème : Fonctions de hachage déterministes produisent toujours le même résultat pour le même mot de passe.

Risques :

- Détection de mots de passe identiques
- Attaques par dictionnaire offline (codebooks pré-calculés)
- Rainbow tables

Solution UNIX : Salt

- Ajout d'un élément pseudo-aléatoire de 12 bits (salt) avant hachage
- Salt différent pour chaque utilisateur
- 4096 possibilités (2^{12}) pour chaque mot de passe

Avantages :

- Empêche détection des duplications
- Codebooks pré-calculés deviennent inefficaces
- Chaque mot de passe nécessite 4096 entrées dans le dictionnaire

Implémentation UNIX :

- Fichier `/etc/passwd` accessible globalement
- Format : `username:hash(salt+password):uid:gid:...`
- Hash basé sur DES modifié (25 itérations)
- Salt stocké en clair (2 premiers caractères du hash)

Exemple :

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

Limitations :

- Protection efficace contre dictionnaires pré-calculés
- Attaques online limitées par le système (nombre d'essais)
- Attaques offline possibles si fichier compromis

Texte original

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 dictionnary 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.

Le schéma de fonctionnement utilise DES avec 25 itérations, le password comme clé, et le salt pour modifier les E-boxes. Le résultat final de 64 bits est converti en 11 caractères ASCII.
La sensibilisation des utilisateurs (ne pas visiter des sites douteux) diminue l'efficacité de cette technique dans la transmission de malware.
Les attaques dictionnaire sont normalement moins efficaces **online** car les systèmes d'exploitation limitent le nombre d'essais infructueux d'authentification.

Révision rapide

UNIX salt : 12 bits aléatoires ajoutés au password avant hash.
4096 variations possibles.
Empêche codebooks pré-calculés et détection duplications.