HAI709I - Fondements cryptographiques de la sécurité

Cours 7 - Fonctions de hachage

Rocco Mora

3 Novembre, 2025

Université de Montpellier – Faculté des Sciences
M1 informatique, parcours Algo, IASD, Imagine

Fonctions de hachage pour les structures de données

Fonction de hachage : [🇬🇧 hash function] une fonction qui prend des entrées d'une certaine longueur et les compresse en sorties courtes et de longueur fixe.

Utilisation principale dans les structures de données : permet de construire des tables de hachage qui permettent un temps de recherche $\mathcal{O}(1)$ lors du stockage d'un ensemble d'éléments

1. Stockez l'élément x dans la ligne $H(x)$ d'une table de taille N, où N est la taille du range de l'entrée.
2. Pour récupérer x, calculez $H(x)$ et recherchez dans cette ligne de la table les éléments qui y sont stockés.

Deux entrées x, x' entrent en collision si $H(x) = H(x')$.

$\rightarrow$ lorsqu'une collision se produit, deux éléments sont stockés dans la même ligne, ce qui augmente le temps de recherche.

Ainsi, une fonction de hachage H est bonne si

- elle ne présente que peu de collisions
- lorsque l'ensemble des éléments hachés est choisi indépendamment de H.

Fonctions de hachage résistantes aux collisions :

- elles doivent éviter les collisions
- lorsqu'un adversaire peut sélectionner des éléments hachés.

Les fonctions de hachage résistantes aux collisions sont beaucoup plus difficiles à concevoir

Fonction de hachage

Fonction de hachage

Une fonction de hachage avec une longueur de sortie $\ell(n)$ est une paire d'algorithmes polynomiales (Gen, H) satisfaisant les conditions suivantes :

- Gen est un algorithme probabiliste qui prend en entrée un paramètre de sécurité 1^n et produit une clé s. Nous supposons que n est implicite dans s.
- H est un algorithme déterministe qui prend en entrée une clé s et une chaîne $x \in \{0,1\}^*$ et produit une chaîne $H^s(x) \in \{0,1\}^{\ell(n)}$.

Si H^s n'est défini que pour les entrées x de longueur $\ell'(n) > \ell(n)$, alors (Gen, H) est une fonction de hachage à longueur fixe pour les entrées de longueur $\ell'(n)$. Dans ce cas, H est également appelé fonction de compression.

⚠ H^s est une fonction à clé.

Fonctions de hachage résistantes aux collisions

L'expérience de recherche de collision $\texttt{Hash} - \texttt{coll}_{\mathcal{A},\mathcal{H}}(n)$

Soit $\mathcal{H} = (\text{Gen}, H)$ une fonction de hachage.

1. Une clé s est générée en exécutant $\text{Gen}(1^n)$.
2. L'adversaire $\mathcal{A}$ reçoit s et produit x, x'. Si $\mathcal{H}$ est de longueur fixe pour des entrées de longueur $\ell'(n)$, alors $x, x' \in \{0,1\}^{\ell'(n)}$.
3. Le résultat de l'expérience est 1 si et seulement si $x \neq x'$ et $H^s(x) = H^s(x')$. Dans ce cas, on dit que $\mathcal{A}$ a trouvé une collision.

Fonction de hachage résistante aux collisions

Une fonction de hachage $\mathcal{H}$ est résistante aux collisions [🇬🇧 collision resistant] si, pour tout adversaire PPT $\mathcal{A}$, il existe une fonction négligeable $\texttt{negl}$ telle que

$$\Pr(\texttt{Hash} - \texttt{coll}_{\mathcal{A},\mathcal{H}}(n) = 1) \leq \texttt{negl}(n).$$

Fonctions de compression

Construction de Davis-Meyer : Étant donné un chiffrement par blocs F avec une longueur de clé de n bits et une longueur de bloc de ℓ bits, définissons la fonction de compression $h\colon \{0,1\}^{n+\ell} \to \{0,1\}^{\ell}$

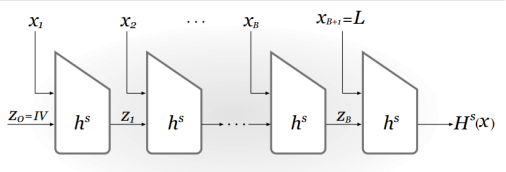
$$\boxed{h(k,x) = F_k(x) \oplus x.}$$

- On ne sait pas s'il est possible de prouver la résistance aux collisions en supposant simplement que F est une permutation pseudo-aléatoire forte, mais
- preuve de résistance aux collisions si F est modélisé comme un chiffrement idéal (simile au modèle d'oracle aléatoire)
- Comment convertir une fonction de compression avec une entrées courte pour entrée de longueur fixe en fonction de hachage pour entrées de longueur arbitraire ?

Transformation de Merkle-Damgård

Soit (Gen, h) une fonction de compression avec une longueur d'entrée $n + n' \geq 2n$ et une longueur de sortie n. Fixons $\ell \leq n'$ et $IV \in \{0,1\}^n$. Construir une fonction de hachage (Gen, H) où Gen reste inchangé et H, en entrée, une clé s et une chaîne $x \in \{0,1\}^*$ de longueur $L < 2^\ell$, est construite de la manière suivante :

1. Ajouter un 1 à x, suivi d'un nombre suffisant de zéros pour que la longueur de la chaîne résultante soit inférieure de ℓ à un multiple de n'. Ajouter ensuite L, codé sous forme de chaîne de ℓ bits. Decomposer la chaîne résultante comme une séquence de blocs de n' bits $x_1, \ldots, x_B$.
2. Définir $z_0 := IV$
3. Calculer $z_i := h^s(z_{i-1}||x_i), i = 1, \ldots, B$.
4. Afficher z_B.



Theorem

Si (Gen, h) est résistante aux collisions ,alors (Gen, H) est également résistante aux collisions.

Fonctions de hachage standardisées

Construction de Davis-Meyer + la transformation de Merkle-Damgård :

- **MD5** : conçue en 1991, aujourd'hui complètement compromise
- **SHA-1** : fait partie des Secure Hash Algorithms (SHA) normalisés par le NIST. Conçue en 1995, nombreuses faiblesses théoriques → pas recommandée
- **Famille de hachage SHA-2** : introduite en 2001, elle se compose de 2 fonctions de hachage (SHA-256 et SHA-512 en fonction de la longueur de sortie). Actuellement recommandée lorsqu'une résistance aux collisions est nécessaire.

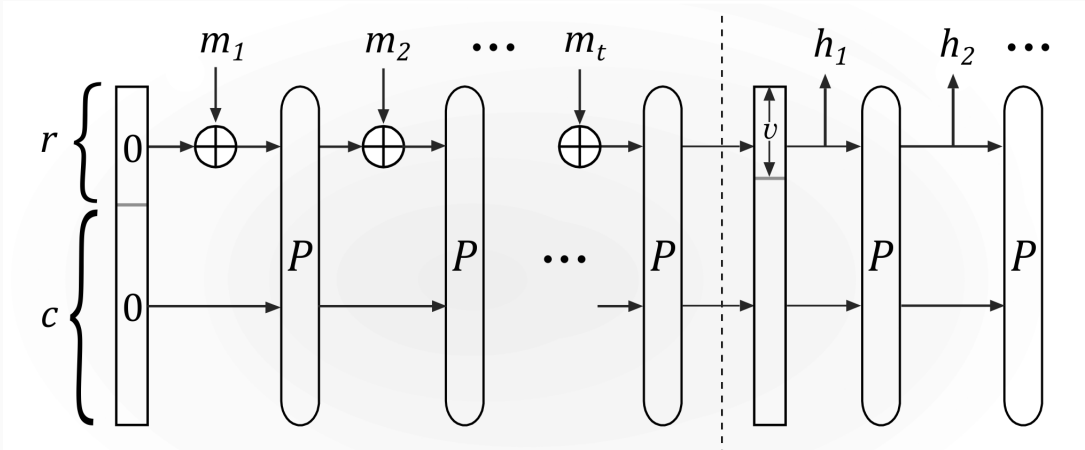
SHA-3 : Concours lancé en 2007 par le NIST pour remplacer MD5 et SHA-1.

- **Famille Keccak** sélectionnée comme gagnante parmi 51 candidats.
  - Très différente de SHA-1 et SHA-2, pas de transformation de Merkle-Damgård
  - Basée sur une permutation sans clé P avec une longueur de bloc de 1600 bits
  - P utilisée pour construire une fonction de hachage via la construction de l'éponge
  - dans le modèle de permutation aléatoire (analogue à l'oracle aléatoire), la fonction de hachage peut être prouvée comme étant résistante aux collisions

La construction éponge [🇬🇧 sponge construction]

Fixons une permutation $P\colon \{0,1\}^\ell \to \{0,1\}^\ell$, et posons $r, c, v, \lambda \geq 1$ tels que $r + c = \ell$ et $v \leq \ell$. Sur l'entrée $\hat{m} \in \{0,1\}^*$, la fonction de hachage H effectue :

- **Remplissage [🇬🇧 Padding] :** Ajouter un 1 à $\hat{m}$, suivi de zéros, de sorte que la chaîne résultante soit un multiple de r. Decomposer la chaîne comme une séquence de blocs de r bits $m_1, \ldots, m_t$.
- **Phase d'absorption [🇬🇧 Absorbing phase] :** Définir $y_0 = 0^\ell$. Ensuite, pour $i = 1, \ldots, t$, faire
  - $x_i := y_{i-1} \oplus (m_i || 0^c)$;
  - $y_i := P(x_i)$.
- **Phase d'essorage.[🇬🇧 Squeezing phase] :** Définir $y_1^* := y_t$ et soit h_1 les premiers v bits de y_1^*. Ensuite, pour $i = 2, \ldots, \lambda$, calculer :
  - $y_i^* := P(y_{i-1}^*)$;
  - Soit h_i les premiers v bits de z_i^*.
- Sortie $h_1 || \ldots || h_\lambda$.


m_1
m_2
$\cdots$
m_t
h_1
h_2
$\cdots$
r
c
0
0
P
P
$\cdots$
P
v
P
P

Hachage et MAC

Construisons un MAC pour des messages de longueur arbitraire à partir d'une fonction de hachage résistante aux collisions.

Hachage et MAC [🇬🇧 Hash and MAC]

Soit $\Pi = (\mathsf{Mac}, \mathsf{Vrfy})$ un MAC pour des messages de longueur $\ell(n)$, et soit $\mathcal{H} = (\mathsf{Gen}_H, H)$ une fonction de hachage avec une sortie de longueur $\ell(n)$. Construisons un MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ pour des messages de longueur arbitraire comme suit :

- Gen' : sur l'entrée 1^n, choisir uniformément $k \in \{0,1\}^n$ et exécuter $\mathsf{Gen}_H(1^n)$ pour obtenir s, puis afficher la clé (k, s).
- Mac' : en entrée (k, s) et un message $m \in \{0,1\}^*$, afficher $t \leftarrow \mathsf{Mac}_k(H^s(m))$.
- Vrfy' : en entrée une clé (k, s), un message $m \in \{0,1\}^*$ et une étiquette t, afficher 1 si et seulement si $\mathsf{Vrfy}_k(H^s(m), t) = 1$.

Théorème

Si Π est un MAC sûr pour les messages de longueur $\ell(n)$ et que $\mathcal{H}$ est résistant aux collisions, alors la construction précédente est un MAC sûr pour des messages de longueur arbitraire.

Inconvénients de l'approche Hash and MAC :

- nécessite de 2 primitives cryptographiques : fonction de hachage + chiffr. par bloc
- la longueur de sortie des 2 primitives est souvent différente

Pour aller plus loin : HMAC est un construction qui utilise la transformation de Merkle-Damgård et donne un MAC à partir de n'importe quelle fonction de hachage, par exemple SHA2 ou SHA3.

Attaques génériques sur les fonctions de hachage

Attaques par force brute $\rightarrow$ tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Attaques génériques sur les fonctions de hachage

Attaques par force brute $\rightarrow$ tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Réponse : non, à cause du paradoxe des anniversaires !

Attaques génériques sur les fonctions de hachage

Attaques par force brute $\rightarrow$ tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Réponse : non, à cause du paradoxe des anniversaires !

Exemple : Il y a n personnes dans une salle.

Attaques génériques sur les fonctions de hachage

Attaques par force brute → tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Réponse : non, à cause du paradoxe des anniversaires !

Exemple : Il y a n personnes dans une salle.

Soit $n = 50$. Quelle est la probabilité que toutes ces personnes soient nées à des dates différentes de l'année ?

$$\begin{aligned} & \frac{365}{365} \cdot \frac{364}{365} \cdot \dots \cdot \frac{365 - n + 1}{365} \\ = & \frac{365}{365} \cdot \frac{364}{365} \cdot \dots \cdot \frac{365 - 50 + 1}{365} \\ \approx & 0.03. \end{aligned}$$

Attaques génériques sur les fonctions de hachage

Attaques par force brute $\rightarrow$ tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Réponse : non, à cause du paradoxe des anniversaires !

Exemple : Il y a n personnes dans une salle.

Et avec $n = 100$?

$$\frac{365}{365} \cdot \frac{364}{365} \cdot \dots \cdot \frac{365 - 100 + 1}{365} < 10^{-6}.$$

Attaques génériques sur les fonctions de hachage

Attaques par force brute $\rightarrow$ tout schéma assurant la sécurité contre un attaquant en temps 2^n nécessite des clés secrètes d'une longueur d'au moins n bits.

Est-ce que n bits suffisent pour une fonction de hachage bien conçue ?

Réponse : non, à cause du paradoxe des anniversaires !

Exemple : Il y a n personnes dans une salle.

Quelle est la valeur minimale de n telle que la probabilité que deux personnes soient nées le même jour de l'année soit supérieure à 50 % ?

$$1 - \frac{365}{365} \cdot \frac{364}{365} \cdot \dots \cdot \frac{365 - 22 + 1}{365} \approx 0.476$$

$$1 - \frac{365}{365} \cdot \frac{364}{365} \cdot \dots \cdot \frac{365 - 23 + 1}{365} \approx 0.507$$

Réponse : 23

Attaque des anniversaires

Soit $H : \{0,1\}^* \to \{0,1\}^\ell$ une fonction de hachage.

Principe des tiroirs/du pigeonnier : si $|S| = n$ et $x_1, \ldots, x_{n+1} \in S$, au moins deux éléments sont égaux.

$\Rightarrow$ l'évaluation de H sur $2^\ell + 1$ éléments distincts conduit à une collision.

Attaque des anniversaires : [🇬🇧 Birthday attack] (pour simplifier, nous considérons H comme une fonction aléatoire)

Choisissez des éléments distincts uniformes $x_1, \ldots, x_q \in \{0,1\}^*$, avec $q = \Theta(2^{\ell/2})$.
L'évaluation de H sur ces éléments conduit à une collision avec une probabilité de $1/2$.

- $+$ bien meilleur que la recherche par force brute
- $-$ nécessite une grande quantité de mémoire : $\Theta(q) = \Theta(2^{\ell/2})$

Attaque des anniversaires en place

A small-space birthday attack

Input: A hash function $H : \{0,1\}^* \to \{0,1\}^\ell$
Output: Distinct x, x' with $H(x) = H(x')$

$x_0 \leftarrow \{0,1\}^{\ell+1}$
$x' := x := x_0$
for $i = 1, 2, \ldots$ **do**:
  $x := H(x)$
  $x' := H(H(x'))$
  // now $x = H^{(i)}(x_0)$ and $x' = H^{(2i)}(x_0)$
  if $x = x'$ **break**
$x' := x$, $x := x_0$
for $j = 1$ to i:
  if $H(x) = H(x')$ **return** x, x' and **halt**
  else $x := H(x)$, $x' := H(x')$
  // now $x = H^{(j)}(x_0)$ and $x' = H^{(i+j)}(x_0)$

L'algorithme nécessite un espace de mémoire constant

Autres applications des fonctions de hachage (1/3)

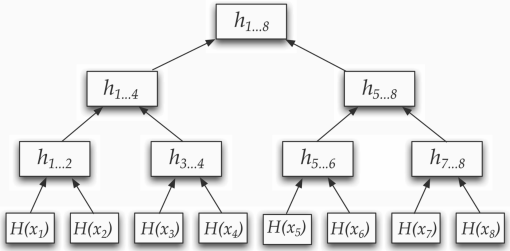
Le hachage $H(x)$ peut être utilisé comme empreinte digitale d'un fichier x.

- **Empreinte digitale des virus :** les antivirus peuvent identifier les virus plus rapidement en comparant les hachages
- **Déduplication :** lorsque plusieurs utilisateurs souhaitent stocker le même fichier, il suffit de le télécharger et de le stocker une seule fois, puis d'utiliser le hachage pour éliminer les copies en double (par exemple dans les services de collusion)
- **Partage de fichiers peer-to-peer (P2P) :** les serveurs stockant différents fichiers peuvent faire connaître leur contenu en diffusant leurs hachages.

Autres applications des fonctions de hachage (2/3)

Scénario : Une cliente télécharge un fichier sur un serveur et, lorsqu'elle souhaite le récupérer, elle vérifie qu'il correspond à l'empreinte digitale. Que se passe-t-il s'il y a beaucoup de fichiers ?

- Hachage de tous les fichiers indépendamment $H(x_1), \ldots, H(x_t)$
  - \+ La surcharge de communication est constante.
  - − Le stockage du client augmente linéairement en fonction de t.
- Hachage de tous les fichiers ensemble $H(x_1, \ldots, x_t)$
  - − pour vérifier un fichier, le client doit récupérer tous les fichiers, c'est-à-dire que la surcharge de communication est linéaire en t
  - \+ le stockage du client est constant
- Solution de compromis : arbres de Merkle [🇬🇧 Merkle tree], c-à-d un arbre binaire de profondeur $\log(t)$, avec t fixè, dont les feuilles sont des hachages de fichiers et les nœuds internes des hachages des 2 enfants.



Dans l'image ci-dessus, le client calcule la racine $h_{1...8} = \mathcal{MT}_8(x_1, \ldots, x_8)$ et télécharge $x_1, \ldots, x_8$ sur le serveur. S'il souhaite récupérer x_3 :

1. Le serveur envoie x_3 avec $h_4, h_{1\dot{2}}$ et $h_{5...8}$.
2. Le client calcule $h'_3 = H(x_3), h'_{3...4} = H(h_{3\mathrm{J}}, h_4), h'_{1...4} = H(h_{1...2}, h'_{3...4}), h'_{1...8} = H(h'_{1...4}, h_{5...8})$ et vérifie si $h'_{1...8} = h_{1...8}$.

+ la surcharge de communication est logarithmique en t
+ le stockage du client est constant
+ si (Gen_H, H) est résistant aux collisions, alors $(\mathrm{Gen}_H, \mathcal{MT}_t)$ est résistant aux collisions pour tout t fixe.

Autres applications des fonctions de hachage (3/3)

Le hachage $H(pw)$ peut être stocké dans un fichier à la place du mot de passe pw

- Le système d'exploitation vérifie que le hachage du mot de passe saisi par l'utilisateur est égal à $H(pw)$.
- Même si un adversaire vole l'ordinateur portable et lit $H(pw)$, il ne récupère pas pw
- L'ensemble de toutes les entrée possibles doit rester suffisamment grand, sinon l'adversaire peut énumérer tous les mots de passe possibles et vérifier l'égalité des hachages.
- Le preprocessing peut être utilisé pour calculer des tables de hachage et récupérer des milliers de mots de passe en cas de violation du serveur
  $\rightarrow$ atténuation possible : ajouter du sel [🇬🇧 salt] s qui dépend de l'utilisateur, et stocker $(s, H(s, pw))$.