MODULE ID12 : INTRODUCTION À LA CRYPTOLOGIE
COURS MAGISTRAL 2

PAUL ZIMMERMANN

1. Hachage et intégrité des données

Référence : chapitre 9 de [1].

Les fonctions de hachage sont utiles pour garantir l'intégrité des données (calcul de *checksum*, exemple commandes `cksum` et `md5sum`), pour la signature.

Définition : Une fonction de hachage prend en entrée une donnée $d \in \{0,1\}^*$ et renvoie $0 \leq h < 2^n$ (n fixé).

Propriétés de base : (i) compression : pour toute donnée x de n'importe quelle taille, on peut calculer $h(x)$ de taille fixe ; (ii) efficacité : $h(x)$ est facile à calculer (typiquement de complexité linéaire en la taille de x).

Propriétés cryptographiques :

(1) difficile à inverser (à sens unique) : étant donné y, il est difficile de trouver x tel que $h(x) = y$. Sinon, permet de trouver un message en clair correspondant à une signature donnée. Idéal 2^n. (*preimage resistance* ou *one way* en anglais).

(2) étant donné x, difficile de trouver x' tel que $h(x) = h(x')$ (*2nd-preimage resistance* ou *weak collision resistance* en anglais). Idéal 2^n.

(3) résistante aux collisions : difficile de trouver x et x' tels que $h(x) = h(x')$. Sinon, permet de trouver deux messages ayant la même signature (*collision resistance* ou *strong collision resistance* en anglais). Idéal $2^{n/2}$ (paradoxe des anniversaires).

Le paradoxe des anniversaires. Soit une valeur aléatoire pouvant prendre d valeurs différentes. Après n tirages indépendants, la probabilité d'avoir tiré n valeurs différentes est :

$$p_{d,n} = 1(1 - 1/d)(1 - 2/d) \cdots (1 - (n-1)/d).$$

Par exemple $p_{365,23} \approx 0.49$: lorsque 23 personnes ou plus sont réunies dans une salle, la probabilité que deux d'entre elles aient même date anniversaire est plus grande que 1/2. (Pour le même jour du mois, on a $p_{31,7} \approx 0.48$.)

Exemple. Soit $h(x) = x^2 \bmod p$ pour p premier. Ce n'est pas une fonction à sens unique car le calcul de racine carrée modulo p est facile. Pour $n = pq$ avec p, q premiers, $h(x) = x^2 \bmod n$ est à sens unique, car le calcul de racine carrée modulo n est réputé difficile. Par contre $h(x)$ n'est pas résistant aux collisions (faibles ou fortes), car $h(x) = h(-x)$.

Exemple. Pour p et q premiers, la fonction $f(p, q) = p \cdot q$ est à sens unique, car calculer $n = p \cdot q$ à partir de p et q est facile, par contre retouver p et q à partir de n est difficile.

Relations entre les propriétés : la résistance forte aux collisions entraîne la résistance faible [preuve]. Par contre la résistance aux collisions ne garantit pas la difficulté de l'inversion.

Date: 11 octobre 2005. Email : `zimmerma@loria.fr`.

Soit $g(x)$ une fonction de hachage résistante aux collisions, et produisant des mots de n bits. Définissons h par :

$$h(x) = \begin{cases} 1x & \text{si } x \text{ a taille } n \\ 0g(x) & \text{sinon.} \end{cases}$$

Alors $h(x)$ est une fonction de hachage de $n+1$ bits, résistante aux collisions, mais facile à inverser.

Deux classes de fonctions de hachage :

- sans clé : servent à garantir uniquement l'intégrité du message (*modification detection code* ou MDC en anglais). Génériques (exemples Merkle, Matyas-Meyer-Oseas) ou ad-hoc (exemples MD4, MD5, SHA-1) ;
- avec clé : garantissent à la fois l'intégrité du message et l'identité de l'expéditeur (*message authentication code* ou MAC en anglais). Exemples : génériques (CBC-MAC) ou ad-hoc (MAA, MD5-MAC).

Note : SHA-1 a été « cassé » en février 2005 par Wang, Yin et Yu, qui ont montré que des collisions pouvaient être trouvées en 2^{69} essais au lieu de 2^{80}.

n est la taille de l'entrée (blocs), m la taille de la sortie.

fonction	n	m	inverse	collision
Matyas-Meyer-Oseas	n	m	2^n	$2^{n/2}$
MDC-2	64	128	2^{83}	2^{55}
MD4	512	128	2^{128}	2^{20}
MD5	512	128	2^{128}	2^{64}
SHA-1	512	160	2^{160}	2^{80}
RIPEMD-160	512	160	2^{160}	2^{80}

1.1. **Fonctions de hachage sans clé.** Algorithme de Merkle : transforme une fonction de compression $f : \{0,1\}^{n+r} \to \{0,1\}^n$ résistante aux collisions, en une fonction de hachage $h : \{0,1\}^* \to \{0,1\}^n$, résistante aux collisions.

> Algorithme de Merkle :
> Couper entrée x de b bits en $x_1 x_2 \dots x_t$, chaque x_i étant de taille r, ajoutant des 0 à x_t si besoin ;
> Ajouter $x_{t+1} = b$ (cadré à droite)
> Calculer $H_i = f(H_{i-1} x_i)$ avec $H_0 = 0^n$
> Renvoyer H_{t+1}.

Algorithme générique de Matyas-Meyer-Oseas basé sur une fonction de chiffrement par bloc E_K :

> Algorithme de Matyas-Meyer-Oseas
> Input : une suite de bits x
> Output : une valeur hachée de n bits
> Couper x en blocs de n bits $x_1 x_2 \dots x_t$ (compléter si besoin)
> $H_0 = \text{IV}$, $H_i = E_{H_{i-1}}(x_i) \oplus x_i$
> Renvoyer H_t.

1.2. **Fonctions de hachage avec clé.** Essentiellement utilisée pour l'authentification de message.

Algorithme CBC-MAC (chiffrement par bloc, IV = 0).
Input : donnée x, chiffrement par bloc E, clé secrète k
Output : valeur de hachage sur n bits pour x
Couper x en $x_1 \dots x_t$ (compléter si besoin)
$H_1 = E_k(x_1)$, $H_i = E_k(H_{i-1} \oplus x_i)$
Retourner H_t.

1.3. **Intégrité des données.** Problème : comment garantir l'intégrité des données sur un canal non sûr ?

Trois méthodes essentiellement :

(1) utiliser une fonction de hachage avec clé (MAC) : on transmet le message en clair et la signature ;

(2) utiliser une fonction de hachage sans clé et une fonction de chiffrement : on chiffre le message en clair et le *checksum*, et on transmet le contenu chiffré ;

(3) si on dispose d'un canal sûr, on peut transmettre le message en clair sur le canal non sûr, et le *checksum* sur le canal sûr (exemple téléphone).

Trois types d'intégrité :
– intégrité des données. Exemple : un fichier binaire.
– lien message-expéditeur : implique l'intégrité des données.
– identification d'une transaction : identifie en plus de manière unique la date du message (empêche le rejeu). Il suffit en fait d'avoir un procédé d'intégrité message-expéditeur, avec des paramètres dépendant du temps.

Attaque de Yuval (paradoxe des anniversaires).
Input : message original x_1, message corrompu x_2, fonction h de hachage sur m bits
Output : x_1', x_2' modifications mineures de x_1, x_2, avec $h(x_1') = h(x_2')$
Calculer $2^{m/2}$ modifications mineures x_1' de x_1
Calculer les $h(x_1')$ et les stocker dans une table T
Faire des modifications mineures x_2' de x_2, jusqu'à ce qu'un $h(x_2')$ soit dans T
Renvoyer alors x_2' et le x_1' correspondant.

2. Générateurs pseudo-aléatoires

Référence : chapitre 5 de [1].

Définition : un *générateur aléatoire* de bits est un procédé produisant une suite de bits à la fois indépendants et non biaisés ($\Pr(0) = \Pr(1) = \frac{1}{2}$).

Définition : un générateur *pseudo-aléatoire* de bits est un algorithme déterministe qui, étant donnée une suite aléatoire de k bits (souche ou *seed* en anglais), produit une suite de l bits qui semblent aléatoires.

Générateur linéaire congruentiel : souche x_0,

$$x_n = ax_{n-1} + b \bmod m.$$

Exemple dans POSIX :

```
/* RAND_MAX assumed to be 32767 */
int myrand(void) {
```

```
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

2.1. **Générateurs vraiment aléatoires.**

(1) matériels : temps entre deux émissions de particules radioactives, sortie son d'un microphone, ... ;

(2) logiciels : horloge, temps entre deux touches ou mouvements de souris, charge machine ou réseau ;

(3) élimination du biais : exemple si un générateur biaisé produit 0 avec $p < \frac{1}{2}$, alors en remplaçant 01 par 0, 10 par 1, et en jetant 00 et 11, on obtient un générateur non biaisé.

2.2. **Générateurs pseudo-aléatoires.**

Algorithme ANSI X9.17
Entrée : souche s de 64 bits, entier m, clé DES k
Sortie : m nombres pseudo-aléatoires de 64 bits $x_1, x_2, \ldots, x_m$
1. Calculer $I = E_k(D)$, où D est un codage 64 bits de la date
2. **for** $i := 1$ **to** m **do**
    $x_i \leftarrow E_k(I \oplus s)$
    $s \leftarrow E_k(x_i \oplus I)$
3. Renvoyer $x_1, x_2, \ldots, x_m$

Algorithm RSA-Random
1. $n = pq$, $\phi = (p-1)(q-1)$, $1 < e < \phi$, $\gcd(e, \phi) = 1$
2. select a random $x_0 \in [1, n-1]$
3. **for** $i := 1$ **to** l **do**
    $x_i \leftarrow x_{i-1}^e \bmod n$
    $z_i \leftarrow$ the least significant bit of x_i
Return $z_1, z_2, \ldots, z_l$

Remarque : on peut prendre $e = 3$, et prendre $c \log\log n$ bits au lieu de 1 bit.

Variante de Micali-Schnorr : génère k bits à la fois (bits de poids faible de x_i), en remplaçant x_i par les r bits de poids fort de $x_{i-1}^e \bmod n$. Produit $(1 - \frac{2}{e}) \log_2 n$ bits à la fois (pour $N = 1024$, produit 341 bits à la fois).

2.2.1. *One-time password.*

Algorithm OneTimePassword (Lamport)
I_1. User A chooses a secret w, a one-way function H, and a constant t
I_2. A transfers $w_0 = H^t(w)$ to B, and B sets $i_A = 1$
A_i. A sends to B the message $A, i, w_i = H^{t-i}(w)$
V_i. B checks that $i = i_A$, $H(w_i) = w_{i-1}$, and $i_A \leftarrow i_A + 1$.

Références

1. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, *Handbook of applied cryptography*, CRC Press, 1997, freely available at `http ://www.cacr.math.uwaterloo.ca/hac/`.