TD9 - Fonctions de Hachage et Cryptanalyse

Responsable : M. Bombar

1 Propriétés de Sécurité des Fonctions de Hachage

1.1 Résistance aux collisions n'implique pas résistance aux préimages

Soit $h : \{0,1\}^\star \to \{0,1\}^n$ une fonction de hachage qu'on suppose **résistante aux collisions**. On construit la fonction de hachage suivante

$$h' : \begin{cases} \{0,1\}^\star & \to \{0,1\}^{n+1} \\ x & \mapsto \begin{cases} 0||x & \text{si } |x| = n. \\ 1||h(x) & \text{sinon.} \end{cases} \end{cases}$$

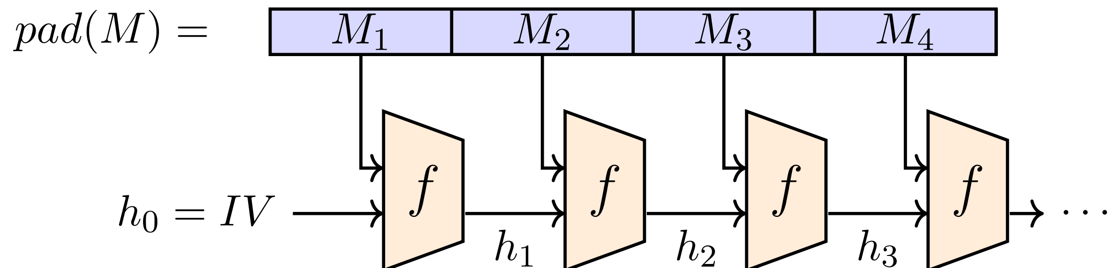
(Q1) Montrez que h' est résistante aux collision.

(Q2) Rappelez la définition formelle de la résistance à la préimage.

(Q3) Montrez que h' n'est pas résistante à la préimage.

1.2 Length-Extension Attack sur la construction de Merkle-Damgård

Soit H une fonction de hachage construite selon la méthode de Merkle-Damgård (par exemple MD5, SHA1, SHA256), dont on rappelle la construction ci-dessous.



(Q4) Soit $h = H(m) \in \{0,1\}^n$ le hash d'un message $m \in \{0,1\}^\star$ considéré comme secret. Soit $\sigma \in \{0,1\}^\star$ un suffixe quelconque (choisi par l'attaquant). Montrez qu'il est possible d'obtenir efficacement le hash d'un message étendu de la forme $m||padding||\sigma$ en temps $O(|\sigma|)$.

Indication : Supposer dans un premier temps que le message est de la bonne longueur, c'est-à-dire qu'il n'y a pas de *padding* interne.

(Q5) On considère un client C et un serveur S ayant partagé une clé secrète sk (par exemple à l'aide d'un protocole d'échange de clés comme celui de Diffie-Hellman). Afin de limiter les attaques de type *Man-in-the-middle* dans la suite du protocole (on suppose que sk n'est pas compromise), on propose de rajouter à chaque message m échangé entre C et S un identifiant supplémentaire de la forme $H(sk||m)$. Ainsi, chaque message échangé entre C et S est maintenant de la forme

Montrez que ce processus n'est absolument pas sécurisé dans le sens où un attaquant peut modifier le message m de sorte à obtenir un paquet valide entre C et S.

2 Fonction de Hachage en Python

Dans ce TD, on va chercher à établir des collisions. Vous pourrez peut-être avoir besoin de générer des bits aléatoires, ce qui peut se faire via la bibliothèque random :

```
sage: import random

sage: random.getrandbits(64)
12346132557762995779

sage: random.getrandbits?
Signature:      random.getrandbits(k, /)
Docstring:      getrandbits(k) -> x.  Generates an int with k random
     bits.
Init docstring: Initialize self.  See help(type(self)) for accurate
    signature.
File:
Type:           builtin_function_or_method
```

La bibliothèque standard de Python offre une interface vers les fonctions de hachage

MD5, SHA-1 ainsi que les différentes variantes de SHA-2 et SHA-3, à travers la bibliothèque `hashlib`. Attention, ces constructions prennent en entrée des objets de type `bytes` et non pas des `str`. En particulier, vous devez utiliser la méthode `.encode()` ou bien directement construire une chaîne d'octets avec le préfixe `b''`. Je vous invite à aller lire la documentation : https://docs.python.org/3/library/hashlib.html pour plus d'information. Voici un exemple d'utilisation :

```
sage: import hashlib
sage: sha2 = hashlib.sha256
sage: sha2("Hello World!")
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent
    call last)
Cell In[3], line 1
----> 1 sha2("Hello World!")

TypeError: Strings must be encoded before hashing

sage: sha2(b'Hello World!').hexdigest()
'7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069'

sage: sha2('Hello World!'.encode()).hexdigest()
'7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069'
```

3 Collision sur Fonction de Hachage Tronquée

Choisissez un préfixe (par exemple votre prénom) et implémentez l'algorithme générique de recherche de collisions s'appuyant sur le paradoxe des anniversaires pour déterminer deux chaînes de caractères s_1 et s_2 commençant toutes deux par ce préfixe, et de sorte que les 32 premiers bits (*i.e.*, les 8 premiers caractères de l'empreinte hexadécimale) de SHA2(s_1) soient égaux aux 32 premiers bits de SHA2(s_2). Par exemple :

```
Collision found!
Input 1: maxime15857573905157511205
Input 2: maxime13871373172309900626

sha256(input1)=
    a4de129026e4f1b46270dc73772a14c26d90c3df19d2a040d347cc154d38c4f8
sha256(input2)=
    a4de1290d4324581554e4804b53f01f95211371a4241386372502d571fc1e06c

SHA256 prefix (first 32 bits): a4de1290
```

(Q6) De combien d'évaluations de la fonction de hachage avez-vous besoin en moyenne ?

(Q7) Quelle est la complexité en mémoire ?

4 Collisions avec Peu de Mémoire

L'algorithme générique de recherche de collisions via le paradoxe des anniversaires peut en réalité être utilisé pour déterminer une collision entre deux listes arbitraires d'éléments distribués aléatoirement dans un ensemble fini quelconque. Il ne prend pas en compte la façon dont ces éléments aléatoires ont été générés. Cependant, lorsqu'ils correspondent à l'évaluation d'une fonction, comme c'est le cas pour les fonctions de hachage, on peut utiliser des méthodes plus spécifiques qui peuvent améliorer l'utilisation de la mémoire (ce qui peut s'avérer particulièrement crucial pour la cryptanalyse).

4.1 Notre fonction de Hachage

Pour simplifier la suite, on va plutôt identifier une empreinte hexadécimale tronquée à N bits à un entier élément de $\{0, \ldots, 2^N - 1\}$:

```
import hashlib
sha2 = haslib.sha256
N=32

def sha2Trunc(x, N):
    fullHash = sha2(str(x).encode()).hexdigest()
    return ZZ(fullHash[:N//4], 16)
```

4.2 Suite des Itérés

Soit une fonction $H : \{0, \ldots, 2^N - 1\} \to \{0, \ldots, 2^N - 1\}$ (par exemple notre fonction `sha2Trunc`). Partant d'un message de départ X_0, on définit la suite $X_{i+1} \overset{\text{def}}{=} H(X_i)$. Puisqu'elle est à valeurs dans un ensemble fini, on rappelle que cette suite est nécessairement périodique à partir d'un certain rang. En particulier, elle est de la forme suivante :

X_0 X_{c-1} $X_{c+\ell-1}$

On note c la longueur de la pré-période $X_0, \ldots, X_{c-1}$, et ℓ la longueur du cycle, de sorte que $X_0, \ldots, X_{c+\ell-1}$ soient tous distincts deux à deux.

4.3 Recherche du cycle par l'algorithme de Floyd

L'algorithme de Floyd est un algorithme générique pour déterminer un cycle dans une suite d'itérés. Il commence par définir une nouvelle suite Y liée à X par la relation

$$Y_i = X_{2i}, \qquad i.e., \quad Y_0 = X_0 \quad \text{et} \quad Y_{i+1} = H(H(Y_i)).$$

Algorithm 1: Algorithme de Détection de Cycle de Floyd

```
1 x ← H(X_0);
2 y ← H(H(X_0));
3 while x != y do
4     x ← H(x);
5     y ← H(H(y));
6 return x
```

(Q8) Implémentez l'algorithme de Floyd pour trouver un élément $x = X_i$ dans le cycle. Partir de X_0 uniformément distribué dans $\{0, \ldots, 2^N - 1\}$.

(Q9) Écrire un algorithme permettant de retrouver la longueur ℓ du cycle contenant x.

(Q10) Implémenter le calcul de collisions : si $c > 0$ et $\ell > 1$ on a

$$H(X_{c-1}) = X_c = X_{c+\ell} = H(X_{c+\ell-1})$$

et par définition $X_{c-1} \neq X_{c+\ell-1}$. Pour cela, partir de X_0 et X_ℓ et calculer X_i et $X_{\ell+i}$ jusqu'à trouver une égalité.

(Q11) En déduire des exemples de collisions surt les N premiers bits de `SHA256` pour $N = 16, 32, 40, 48$.

Remarque 1. *Contrairement au début de ce TP, la mémoire utilisée ici est négligeable. En revanche, le temps est plus difficile à estimer. Vous pouvez utiliser* `cputime()` *pour mesurer le temps d'exécution : pour chronométrer le temps que met le bloc de code à s'exécuter.*

```
timing = cputime()
[...]
cputime(timing)
```

5 Remarque sur la Parallélisation

L'algorithme présenté ici pour la recherche de cycle est fondammentalement séquentiel. En revanche, il existe des méthodes génériques pour les rendre parallélisables. Vous pouvez par exemple chercher méthode des points distingués.

6 Challenge : Collision entre deux messages qui ont du sens

6.1 Explication

L'objectif ici est de déterminer une collision entre deux messages qui ont du sens. Pour cela, on part de l'observation suivante : étant donné un message M, il existe souvent des modifications qui peuvent lui être appliquées sans en changer le sens. Par exemple, on peut changer une espace en espace insécable ou en deux espaces, changer un mot en un synonyme, ou encore remplacer une minuscule par une majuscule. Bien évidemment, si le sens n'est pas changé, certaines modifications altèrent toutefois son apparence, mais en étant créatif on peut faire des choses peu détectables. C'est d'autant plus vrais dans des

fichiers formatés qui peuvent ignorer des pans complets du fichier (par exemple pour crééer des collisions de PDF). À présent, à partir de t positions pour des modifications possibles sur un message M, on en déduit un espace de 2^t messages différents qui ont le même sens, alors que les hash de ces 2^t messages semblent complètement aléatoires *a priori*.

Pour ce TP, partons de deux messages M et M' écrits en majuscule ayant des sens différents. On construit alors $\mathcal{M} \cup \mathcal{M}'$ où $\mathcal{M}$ est l'ensemble des modifications de M obtenues en basculant en minuscules un sous ensemble fini de caractères de M. L'ensemble $\mathcal{M}'$ est défini de manière analogue par rapport à M'. L'idée est que maintenant ces ensembles sont suffisamment grands pour qu'on puisse trouver un élément de $\mathcal{M}$ et un élément de $\mathcal{M}'$ ayant le même hash (tronqué).

Pour cela, on va considérer une fonction

$$g : \{0, \ldots, 2^N - 1\} \to \mathcal{M} \cup \mathcal{M}'$$

injective (en particulier, il faut que les messages M et M' soient suffisamment longs), et on va chercher une collision en appliquant la méthode précédente à la fonction $H \stackrel{\text{def}}{=} \tilde{h} \circ g$ où

$$\tilde{h} : \mathcal{M} \cup \mathcal{M}' \to \{0, \ldots, 2^N - 1\}$$

correspond à la fonction `sha2Trunc` appliquée sur les éléments de M et M'.

La fonction g. Puisque la fonction g est injective, si $x \neq y$ fournit une collision sur H, alors $g(x) \neq g(y)$ et on en déduit immédiatement une collision sur $\tilde{h}$. Il faut cependant faire attention à ce que les collisions trouvées ne soient pas deux copies de M ou deux copies de M'. On a alors besoin de définir la fonction g avec un critère supplémentaire pour savoir dans quel cas on est.

On peut par exemple définir de la façon suivante. Un élément $\{v \in \{0, \ldots, 2^N - 1\}$ dont l'écriture binaire est $b_0 \ldots b_{N-1}$ va correspondre à une modification de M si $b_0 = 0$ et à une modification de M' si $b_0 = 1$.

- Si $b_0 = 0$, on va lire v et basculer en minuscule la i-ème lettre de M lorsque $b_i = 1$ (pour tout $1 \leq i \leq N - 1$.
- Si $b_0 = 1$, on va appliquer la même transformation mais cette fois à M'.

Ainsi, il faudra bien faire attention que les collisions trouvées correspondent à des entiers de bits de parité différente.

Exemples : On choisit les messages $M \stackrel{\text{def}}{=}$« ALICE A VALIDÉ QUE BOB A UN ACCÈS EN LECURE AUX FICHIERS DANS /HOME/BOB » et $M' \stackrel{\text{def}}{=}$ « ALICE A VALIDÉ QUE BOB A UN ACCÈS ADMINISTRATEUR COMPLET AU SYSTÈME. ». Vous pouvez alors vérifier que si

$\mathbf{m} \stackrel{\text{def}}{=}$ « AlicE a VAlIdÉ Que bob A un aCCès EN leCTUrE aUx FICHIERS DANS /HOME/BOB. »

et

$\mathbf{m}' \stackrel{\text{def}}{=}$ « aLicE A validÉ Que boB A UN aCCÈS AdmInIstRAteuR COMPLET AU SYSTÈME. »,

alors,

$\text{SHA256}(\mathbf{m}) =$ **bb199ddee7**f33d129ff4b4bf8b2bb43d734836df3be57e63993b57a88d3faddb

et

$\text{SHA256}(\mathbf{m}') =$ **bb199ddee7**7d9b1315c2a1f74fa58d4cd4e94d7fc14c83b3902fd39d92e448ba.

En particulier, ils coïncident sur leurs 40 premiers bits.

6.2 À vous de jouer

Choisissez deux messages M et M' avec des sens différents et construisez un modifié de M et un modifié de M' dont les hash tronqués aux 32 premiers bits sont égaux. Vous pouvez tenter d'aller plus loin, mais ça risque de prendre du temps.