Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2021-2022

Rappel : 2e évaluation moodle mardi 19 avril entre 15h et 16h30

Programme : essentiellement cours n° 8 à 11
(ce qui n'interdit pas de poser également des questions sur les cours précédents)

deux (nouvelles) auto-évaluations disponibles, respectivement sur les ABR et sur le hachage

En cas de problème technique...

- soyez patients et ***évitez absolument*** de recharger frénétiquement la page, cela risque essentiellement de provoquer un écroulement total de moodle
- mettez ***un*** message court et précis (***pas une copie d'écran***) sur discord

Pot pourri

I. Hachage – étude de cas : les ensembles en Python

II. Utilisations de fonctions de hachage dans d'autres contextes

~~IIa. Recherche de motif dans un texte~~

IIb. Hachage cryptographique

~~III. ABR : comment assurer un coût logarithmique *au pire*~~

~~+ Questions diverses quand vous voulez~~

Implémentation des ensembles Python

Code disponible sur github :
`https://github.com/python/cpython/blob/main/Objects/setobject.c`
`https:`
`//github.com/python/cpython/blob/main/Include/cpython/setobject.h`

Et pour les dictionnaires :
`https://github.com/python/cpython/blob/main/Objects/dictobject.c`

Résumé des points importants

les entrées sont des couples (clé, valeur hachée de la clé)

but : éviter les recalculs, en particulier dans le cas où la fonction de hachage des objets a été écrite (en python) par l'utilisateur

trois types d'entrées facilement discernables : *unused* (`key=NULL`, `hash=0`), *dummy* (`key=dummy`, `hash=-1`), *active* (`hash!=-1`)

la table stocke à la fois le nombre de clés (`used`) et le nombre total de cases non vides (`fill`)

but : seules les vraies cases vides font des « barrières » efficaces lors des recherches infructueuses, les cases marquées (`dummy`) se comportent comme les cases occupées ; le bon critère pour déclencher un redimensionnement est donc le cumul `fill`, alors que le bon critère pour décider de la nouvelle taille est le nombre de clés (`used`)

la stratégie de sondage est un mélange de sondage linéaire (pendant au plus `LINEAR_PROBES=9` sondages consécutifs) et de sauts, chaque saut étant contrôlé par une portion différente de la valeur `hash` initiale

but : éviter à la fois de rester coincé dans un long cluster, et de multiplier les sauts nécessitant de changer le contenu des caches du processeur ; + « individualiser » autant que possible les sauts en utilisant suffisamment de bits différents de `hash` pour que deux clés ne suivent pas la même suite de sondages

Résumé des points importants

lors de (la recherche précédant) l'ajout, une variable `freeslot` stocke un pointeur vers une case marquée (`dummy`) si on en rencontre

but : garder le plus possible de cases vides (`unused`) pour éviter les longues suites de sondages – donc préférer lorsque c'est possible le recyclage d'une case `dummy` à l'utilisation d'une case vide

pour les dictionnaires : la table est scindée en deux : une (vraie) table de hachage, « creuse » (taux de remplissage limité à $\frac{2}{3}$) contenant uniquement des entiers, correspondant à des indices dans une 2^e table, « dense », contenant les triplets (clé, valeur, clé hachée)

but : gagner de la place, puisque les cases « gâchées » de cette manière ne font que la taille d'un entier et non celle d'un triplet

avantage collatéral : le redimensionnement est plus simple et plus efficace, on peut se contenter de jeter l'ancienne table d'indices et parcourir la table (dense) des entrées

Utilisations de fonctions de hachage dans d'autres contextes

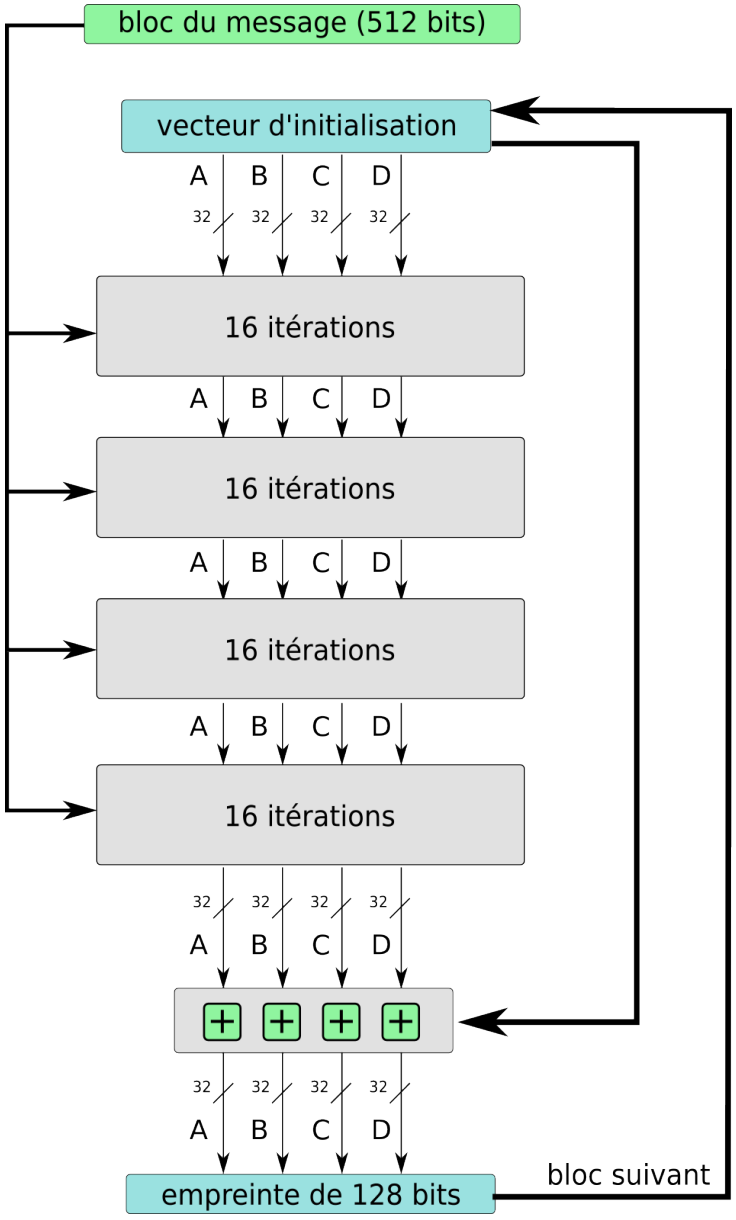
idée : utiliser une fonction qui disperse les données similaires pour coder des données secrètes

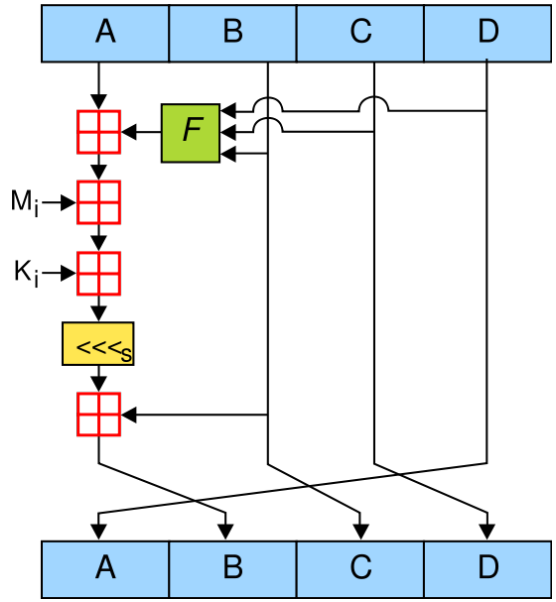
Exemple

- gestion des mots de passe et stockage dans `/etc/shadow`
- vérification de l'intégrité d'un fichier (après un transfert notamment)

$\Longrightarrow$ impératif de sécurité : il doit être difficile d'inverser la fonction – que ce soit pour retrouver la donnée d'origine ou « seulement » trouver une collision

Exemple : MD5





source : Wikipedia

Types d'attaque

attaque *brute-force* : pour déterminer un mot de passe correspondant à une empreinte (ie une valeur hachée) donnée, calculer les empreintes de tous les mots de passe possibles, et s'arrêter quand on tombe sur la bonne

cela pose un problème de complexité – à la fois temporelle et spatiale : si on considère par exemple des mots de passe de 8 lettres minuscules, le nombre de mots de passe possible est de l'ordre de 200 milliards ; on peut envisager de faire (assez rapidement) 200 milliards de calculs, voire de stocker 200 milliards de résultats ; mais si on autorise également les majuscules et les chiffres, on obtient 200 000 milliards de mots de passe possibles… ou bien en passant à 12 minuscules, 100 millions de milliards… le temps nécessaire pour craquer un mot de passe « from scratch » devient rédhibitoire, tout comme le stockage d'un dictionnaire complet (empreinte, mot de passe) calculé tranquillement à l'avance

attaque par tables arc-en-ciel (*rainbow tables*) : compromis temps-espace consistant à précalculer des couples de mots de passe à partir desquels on peut retrouver assez rapidement un mot de passe ayant une empreinte donnée (pas forcément « le bon » mot de passe, mais un mot de passe permettant de passer le test de l'empreinte)

Principe : puisqu'il est difficile/impossible d'inverser h, on contourne le problème : on choisit une (en fait, des) fonction(s) R allant de l'ensemble des empreintes vers l'ensemble des mots de passe possibles, puis on calcule des (beaucoup de) chaînes de la forme suivante (avec ℓ fixé) :

$$m_0 \overset{h}{\longmapsto} e_1 \overset{R_1}{\longmapsto} m_1 \overset{h}{\longmapsto} e_2 \overset{R_2}{\longmapsto} m_2 \ldots \overset{h}{\longmapsto} e_\ell \overset{R_\ell}{\longmapsto} m_\ell$$

on stocke tous les couples (m_0, m_ℓ) correspondants, et pour craquer une empreinte e, on calcule successivement $R_\ell(e)$, $R_{\ell-1}(h(R_\ell(e)))$, ... jusqu'à tomber sur un des m_ℓ stockés ; cela signifie que (*peut-être*) $e = e_i$ pour un certain i de la chaîne correspondante ; on repart alors de m_0 pour calculer m_{i-1} et vérifier si $h(m_{i-1}) = e$.

depuis 2004, on sait engendrer des collisions pour MD5, qui n'est plus considérée comme sûre, même si la falsification de documents n'est pas encore (connue comme) faisable

Solution 1

utiliser des fonctions plus compliquées, produisant un haché plus long (cf famille SHA – SHA1 n'est plus considérée comme sûre, mais (les variantes de) SHA2 l'est (le sont) encore ; SHA3 utilise un principe très différent)

Solution 2

utiliser la technique du ***salage***

Exemple

Linux utilise MD5 avec un *sel* de 8 octets pour gérer les mots de passe

Principe du salage

Contexte : cryptage par une fonction de hachage trop faible, permettant une attaque grâce à une table précalculée

Principe :

- ajouter [1] un grain de *sel* à la donnée *avant* de la hacher :

$$h(x \oplus sel) \text{ est très différent de } h(x)$$

- stocker $h(x \oplus sel)$ et sel : la vérification n'est pas plus compliquée

[1] *i.e.* par exemple concaténer, ou une autre manière de prendre en compte à la fois x et le sel ; le point important est qu'il ne faut pas que l'opération soit inversible

même si l'attaquant obtient toute l'information (empreinte et sel), puis parvient à trouver un mot y dont l'empreinte $h(y)$ est égale à $h(x \oplus sel)$, il y a peu de chances pour qu'il tombe précisément sur un mot terminant par sel, donc y ne lui sert à rien ; donc les attaques doivent être recalculées – à condition que le sel soit différent à chaque fois (*i.e.* au moins pour chaque utilisateur, dans le contexte de la gestion des mots de passe, et même si pour possible pour chaque utilisation)

format de stockage dans `/etc/shadow` : `$i$sel$empreinte`, ou `i` est un code indiquant quelle est la fonction de hachage utilisée (1 pour MD5, 2 pour bcrypt, 5 pour SHA-256...)

le module `crypt` de Python permet de calculer des empreintes « salées » pour diverses fonctions de hachage (du moins sous linux ; sous macOS, il me semble que seule une fonction très très basique est proposée)