Projet de Cryptographie

ESIAL 2A TRS 2012

Groupe P3

Nicolas Bouget, Florent Noyer et Julien Vaubourg

Sommaire

1. Introduction

Le but de ce projet est de mettre en place une fonction de hachage utilisant un chiffrement par bloc, puis d'appliquer différents types d'attaques connues afin d'observer les forces et les faiblesses de ces dernières.

Nous avons réalisé notre projet en Java.

2. Détails Techniques

2.1 Fonction de Hachage

Notre fonction de hachage (cf. Hash.java) est une fonction de chiffrement par bloc CBC utilisant AES (cf. Crypt.java). La clé de chiffrement utilisée est la chaîne que l'on veut hacher et la valeur chiffrée est une graine choisie arbitrairement (ici, nous avons choisi « 123456789 »).

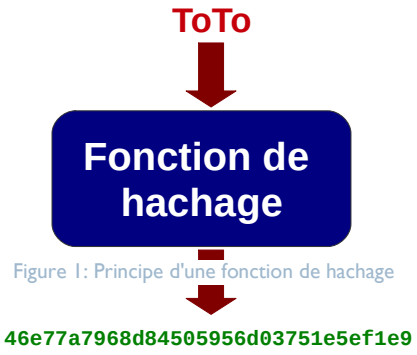

Figure 1: Principe d'une fonction de hachage

Ainsi, inverser notre fonction de chiffrement consiste à retrouver la clé de chiffrement et donc de casser AES. Notre implémentation peut aussi supporter un chiffrement par bloc ECB et les algorithmes DES et RSA.

Remarque : en utilisant DES, on a remarqué beaucoup de collisions, ce qui permettait d'accélérer nos attaques mais cela réduit beaucoup trop la puissance de la fonction de hachage.

Notre système peut hacher des chaînes de 128 bits au maximum étant donné qu'elles sont utilisées comme clé de chiffrement pour AES qui impose cette taille. On peut donc hacher des chaînes de caractères simples d'au plus 16 caractères. Pour le cas où la clé fait moins de 28 bits nous la complétons avec des zéros et à l'inverse, nous la tronquons si elle est trop longue.

Remarque : En Java, les premiers caractères Unicode « simples » sont codés sur 1 octet alors que les suivants sont codés sur 2 octets. Afin de gérer ceci, nous transformons la chaîne en une suite d'octets et nous complétons par des zéros ou tronquons après cette conversion. Ainsi, peu importe les caractères utilisés, nous avons toujours le bon nombre d'octets pour la clé.

En sortie, l'algorithme AES nous rend des mots de 64 bits avec la graine que l'on a choisie. Ainsi, pour rendre le haché lisible, on converti ces bits en caractères hexadécimaux (minuscules et chiffres) et on obtient une chaîne de 32 caractères. On a créé des fonctions qui limitent la taille des chaînes à hacher à 4, 6 ou 8 caractères. Si la chaîne est trop longue, on la tronque.

Enfin, il existe une fonction de hachage spécial qui utilise un double DES afin de pouvoir mettre en œuvre l'attaque *Meet-in-the-middle*.

2.2 Attaque par recherche exhaustive

```
Usage: BruteForce <hash> <maxKeyLength> <method> <threadCount>
```

La recherche exhaustive permet de retrouver n'importe laquelle des chaînes hachées mais cela à un coup exponentiel en temps. Pour des chaînes de 4 caractères avec des majuscules, minuscules non accentuées et des chiffres, on a déjà 14 776 336 (= 62^4) possibilités.

Afin de d'obtenir une attaque qui se termine en un temps raisonnable, on a décidé de prévoir la possibilité d'attaque complète mais de n'utiliser que des versions simplifiées pour nos tests. On a créé des packs de caractères :

- majuscules et minuscules non accentuées + chiffres
- majuscules + minuscules + chiffres + minuscules accentuées fréquentes en français
- majuscules et minuscules accentuées
- majuscules et minuscules accentuées + caractères spéciaux

On va considérer que chaque pack de caractères constitue une base. On va donc incrémenter de 1 (dans la base considérée) à chaque tour de test, puis on hache la chaîne obtenue et on la compare avec celle que l'on veut casser.

Cette méthode facilite la parallélisation, chaque thread ou machine a un intervalle à tester. En effet, on peut diviser l'intervalle total par le nombre de threads/machines. Cependant, on a constaté que le multithread n'était pas une solution, car la machine passe plus de temps à changer de thread que de tester une possibilité. Ainsi, ce partage augmente le temps au lieu de le réduire. Ce problème ne surviendrait pas sur une technique en multi-machines, mais nous n'avons pas implémenté cette solution.

2.3 Attaque par dictionnaire

```
Usage: BruteForceDictionary hash maxKeyLength [option...]
```

On a constitué 6 dictionnaires. Les 3 premiers contiennent respectivement un grand nombre de mots « courants » de moins de 4, 6 ou 8 caractères. Les 3 suivants sont identiques à la différence qu'ils contiennent des prénoms courants. Chaque mot est écrit sous sa forme accentuée et non-accentuée.

On a implémenté 5 types d'attaques par dictionnaire. Tout d'abord, on peut tester tous les mots de chaque dictionnaire pour essayer de trouver une correspondance. Cependant, si on considère que l'on cherche des mots de passe, il arrive souvent que les gens ajoutent des nombres ou concatènent des mots, ainsi on va ensuite tester chaque mot du dictionnaire considéré en lui combinant d'autres mots qui vont permettre d'atteindre la taille maximale ou en ajoutant de part et d'autre des chiffres. La dernière option consiste à tester les mots avec ou sans une première lettre en majuscule.

2.4 Attaque *Meet-in-the-middle*

```
MeetInTheMiddle m = new MeetInTheMiddle(mdp) ;     m.launch(true) ;
```

Le `true` sert à afficher le résultat dans la console (ainsi que le temps de calcul du résultat), et le paramètre `mdp` est le mot de passe que l'on connait.

Pour cette attaque, il a été nécessaire d'utiliser une fonction de hachage particulière : le double DES. La connaissance d'un couple clair/chiffré nous permet ensuite d'effectuer l'attaque.

Tout d'abord on génère l'ensemble des demi-clés possibles. Lorsqu'une demi-clé est générée, le haché du message clair est calculé avec cette demi-clé, puis est stocké dans un *Radix Tree*. Nous avons fait le choix d'utiliser un *Radix Tree* principalement pour rechercher efficacement les hachés à la fin de l'algorithme. Une fois le *Radix Tree* totalement rempli

(i.e. toutes les demi-clés ont été générées), on réutilise les demi-clés générées pour calculer l'inverse du chiffré connu avec la demi-clé générée. On obtient ainsi une liste de candidats pour la clé de chiffrement.

Dans la théorie, l'algorithme fonctionne pour tous les mots de passe de longueur paire (il faut pouvoir séparer le mot de passe en deux parties égales). Mais dans la pratique, les résultats sont trouvés dans un temps acceptable pour les clés de longueur au plus égale à 6. Le calcul pour une clé de longueur 8 demande plus d'une dizaine d'heures. Dans l'objectif de réduire le nombre de possibilités de clés, nous nous sommes restreint à l'étude des mots de passe composés seulement des minuscules, des majuscules et des chiffres.

2.5 Attaque via tables arc-en-ciel

Génération d'un table pour les mots de passe de taille donnée, enregistrée dans le fichier RainbowTableIndice.txt : `RainbowTable rt = new RainbowTable(indice,mdp_size); rt.goTable();`

Recherche du mot de passe de longueur donnée partir d'un haché connu : `RainbowCrack rc = new RainbowCrack(haché,mdp_size); rc.launch();`

L'attaque par tables arc-en-ciel est composée de deux phases. La première est celle de génération des tables, la deuxième étant celle de l'attaque en elle-même.

La principale difficulté rencontrée a été celle du choix des fonctions de réduction. Finalement, voici la famille des fonctions de réductions que l'on a retenu : on découpe le haché donné en n blocs (où n est la longueur du mot de passe cherché). Chaque bloc est ensuite traité de manière similaire, en additionnant la valeur ASCII de chaque caractère composant le bloc. On ajoute ensuite à cela la valeur de l'indice de la fonction de réduction (s'il s'agit de la fonction de réduction r_i on ajoute alors i à la valeur obtenue). De cette manière chaque fonction de réduction est différente. Dans la pratique, nous avons utilisé la même fonction de réduction à chaque itération.

Enfin, par différentes opérations de modulo, on transforme l'entier obtenu en un code ASCII de caractère. La génération des tables se fait ensuite en appliquant un certain nombre de fois (nous avons choisis de faire 30 itérations) la fonction de réduction puis de hachage, sur un grand nombre de mots de passe possibles (générés aléatoirement).

La seconde partie de l'attaque consiste, à partir d'un haché, à retrouver le mot de passe associé.

Ensuite, la recherche du mot de passe associé à un haché se fait de la manière suivante. Si le haché est dans la table arc-en-ciel, alors on part du mot de passe de base et on applique le même procédé que précédemment jusqu'à retrouver le bon haché. Le mot de passe est ainsi le dernier mot de passe trouvé par la fonction de réduction.

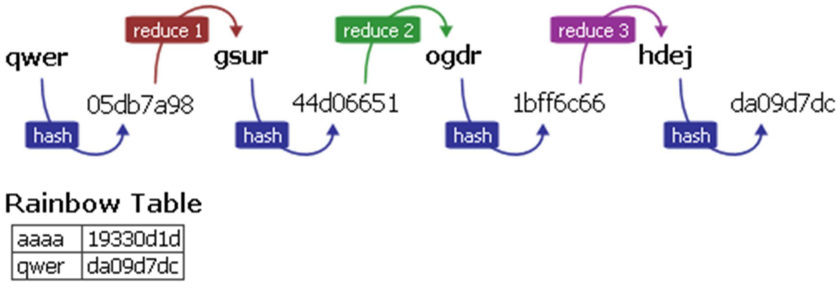

Figure 2: Principe de génération des Rainbow Tables (source : www.mieuxcoder.com)

En revanche si le haché cherché n'est pas dans la table, alors on applique successivement la fonction de réduction et la fonction de hachage au mot de passe. Si le nouveau haché est dans la table, alors on reprend l'étape précédente, sinon on itère.

De cette manière on peut retrouver un mot de passe plus rapidement. Quantitativement, pour stocker 100 million de mots de passe, il suffit de générer une *Rainbow Table* de 100 000 lignes, où chaque ligne est le résultat de 1 000 applications successives de fonctions de réduction. Cela nous permet de stocker une table de 2Mo environ, au lieu d'un dictionnaire équivalent d'environ 2Go. C'est une considérable amélioration de l'attaque par dictionnaire !

Malgré cette amélioration, nous nous sommes restreint à l'étude des mots de passe alphanumériques (majuscules, minuscules et chiffres).

3. Performances et Optimisations

Après quelques tests effectués avec les algorithmes DES et AES, on constate que DES provoque beaucoup plus de collisions sur les longueurs de chaînes que l'on considère. Or ces collisions accélèrent la recherche. On constate que limiter les collisions sur une fonction de hachage est primordial pour assurer une bonne résistance aux attaques. Afin d'optimiser les parcours de solutions, on peut comparer plusieurs hachés à chaque tour de test. La méthode de recherche exhaustive permet cette amélioration.

3.1 Attaque par recherche exhaustive

Pour cette attaque par Brute Force, nous avons construit un pack de caractère avec minuscules, majuscules et chiffres = 62 caractères. On obtient alors :

- 62 mots d'une lettre parcourus en 0,011s, soit 177 µs par mot.
- 3 844 mots de 2 lettres parcourus en 0,137s soit 35 µs par mot.
- 238 328 mots de 3 lettres parcourus en 8,082s soit 34 µs par mot.
- 14 776 336 mots de 4 caractères parcourus en 14m35s soit 60µs par mot.
- 568 00 235 584 mots de 6 caractères parcourus en environ 1 mois (estimation)

3.2 Attaque par dictionnaire

Pour le cas du benchmarks dictionnaire en mode chiffres, avec des mots de maximum 4 caractères : il existe 550 mots de 3 caractères, il n'y a qu'un chiffre à ajouter (avec dix possibilités) et on ne peut le placer que avant ou après : 550*10*2 possibilités. Avec 104 mots de 2 caractères, il faut deux chiffres, et le mot peut être avant, après ou au milieu : 104*10^2*3 possibilités. Enfin, avec 3 mots à 1 caractère, il y a 3 chiffres et 4 positions différentes pour le mot : 3*10^3*4 possibilités. Avec les 2582 mots de 4 caractères, ça fait moins de 100k possibilités.

En suivant le même raisonnement, on trouve plus de 10 millions de possibilités pour les 6 caractères et plus d'un milliard pour les 8 caractères, ce qui donne une tendance exponentielle.

Nos tests ont révélés qu'il fallait 1.425 secondes pour comparer les 3240 mots de 4 chars, 2.495 secondes pour les 33321 mots de 6 chars, et 6.356 secondes pour les 128041 mots de 8 chars. En utilisant les moyennes, on trouve qu'il faut moins d'une minute pour tout tester s'il y a un maximum à 4 caractères, 15 minutes pour 6 caractères, et presque 24 heures pour 8 caractères.

3.3 Meet-in-the-middle

Les performances de cette méthode étant directement liées à la taille de la clé cherchée, un graphique ne serait pas très parlant. En moyenne, il faut 26.4 ms pour retrouver une clé de longueur 2, 525 ms pour une clé de longueur 4, et 33.9 secondes pour une clé de longueur 6. Retrouver une clé de longueur 8 est très long (supérieur à 12h).

3.4 Attaque par tables arc-en-ciel

Nous distinguons ici deux temps de calcul : le temps de génération d'une table de 5000 mots de passe (en itérant 100 fois la fonction de réduction). Une table arc-en-ciel ainsi générée représente 500 000 mots de passe générés aléatoirement.

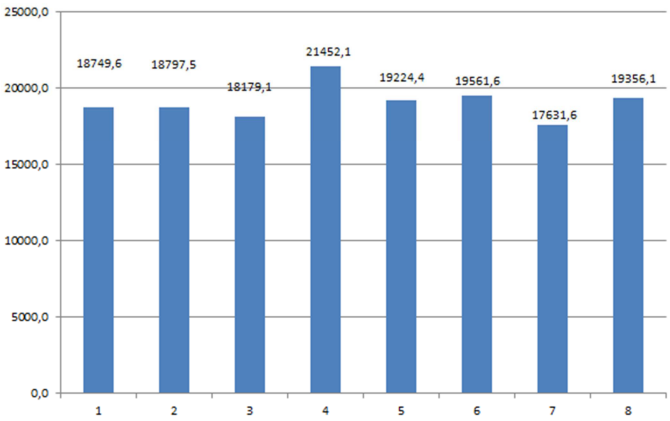

Figure 4 : Temps de génération d'une table (en ms) en fonction de la longueur du mot de passe

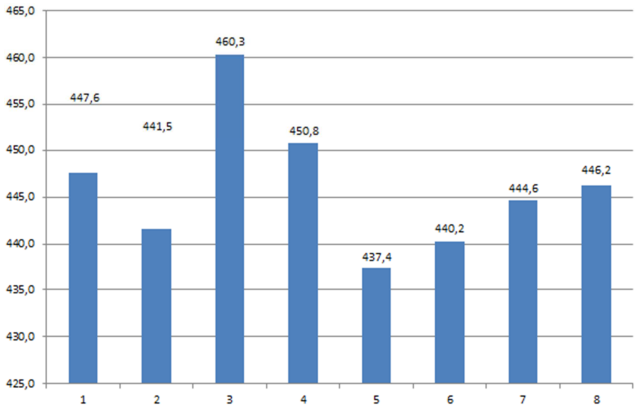

Figure 3 : Temps de résolution (en ms) à partir d'un haché, en fonction de la longueur du mot de passe

Ainsi, en moyenne, une telle table arc-en-ciel est générée en 19.1 secondes, et la résolution du haché sera réalisée en 446 ms. On observe clairement que cette attaque ne dépend pas de la taille du mot de passe, pourvu qu'il existe une table arc-en-ciel correspondant. De plus, le temps de calcul diminue lorsque la taille de la table arc-en-ciel augmente. En effet plus une table contient de mots de passe, moins il faudra de temps pour retrouver celui qui correspond au haché que l'on se donne. Enfin, et c'est sûrement le point le plus important, les performances de ce type d'attaque dépend de la qualité de la fonction de réduction que l'on choisit : moins la fonction de réduction réalise de collisions, plus elle sera efficace. C'est d'ailleurs pour cette raison que la version améliorée de l'attaque par tables arc-en-ciel utilise une nouvelle fonction de réduction pour chaque itération.