Les compromis temps-mémoire et leur utilisation pour casser les mots de passe Windows

Philippe Oechslin

Laboratoire de Securité et de Cryptographie (LASEC)
École Polytechnique Fédérale de Lausanne
Faculté I&C, 1015 Lausanne, Switzerland
philippe.oechslin@epfl.ch

Résumé Les compromis temps-mémoire sont des méthodes qui permettent de réduire le temps d'exécution d'un algorithme en augmentant la quantité de mémoire utilisée. Dans cet article nous présentons différentes variantes de compromis temps-mémoire qui permettent d'accélérer le cassage de mots de passe. Nous expliquons comment configurer un tel compromis pour obtenir le cassage le plus rapide, comment estimer les performances que l'on peut espérer atteindre et nous montrons comment implémenter cette méthode de manière efficace pour casser les mots de passe des systèmes Windows.

1 Introduction

La base de ce travail est une méthode originale publiée en 1980 par Martin Hellman [1] et améliorée par Whitfield Diffie en 1982. Dernièrement nous avons mis au point une variante de cette méthode qui augmente son efficacité d'un ordre de magnitude. Ces méthodes cherchent toutes à résoudre le même problème : Comment inverser rapidement une fonction irréversible. Il peut s'agir par exemple de retrouver la clef qui a servi à chiffrer un texte prédéfini [1] ou, ce qui est notre cas, de retrouver un mot de passe à partir de son empreinte. Dans les deux cas, on peut utiliser une méthode de force brute, qui consiste à essayer toutes les possibilités jusqu'à ce qu'on trouve le bon mot de passe. Une autre approche consisterait à calculer une fois les empreintes de touts les mots de passe possibles et de les stocker, avec les mots de passe, dans un énorme dictionnaire. La force brute est une méthode qui nécessite énormément de temps mais aucune mémoire, alors qu'un dictionnaire complet permet de trouver le mot de passe immédiatement mais à l'aide d'une mémoire énorme. Le but des compromis temps-mémoire est de créer des solutions intermédiaires qui permettent de trouver un mot de passe plus vite qu'avec la force brute en utilisant moins de mémoire qu'un dictionnaire complet.

Dans la suite nous allons d'abord voir le principe général de fonctionnement des compromis temps-mémoire qui nous intéressent. Ensuite nous allons voir de manière plus formelle comment trouver les paramètres optimaux des compromis

[1] *fixed plaintext attack.*

et quelles sont les performances que l'on peut attendre de ces méthodes, en particulier dans le cas des mots de passe Windows. Nous verrons finalement comment dans ce cas précis l'implémentation de la méthode peut être affinée pour obtenir des résultats encore plus performants.

2 Compromis à base de chaînes

Les compromis temps-mémoire sont réalisés à l'aide de chaînes. Pour pouvoir créer des chaînes on définit une fonction de réduction, qui génère un mot de passe arbitraire à partir d'une empreinte. En alternant la fonction de hashage, qui génère une empreinte à partir d'un mot de passe, et la fonction de réduction qui génère un mot de passe à partir d'une empreinte on peut donc générer des chaînes dans lesquels s'alternent mots de passes et empreintes. Pour obtenir un compromis temps-mémoire on génère un nombre m de chaînes de longueur t et on stocke le premier et le dernier élément de chaque chaîne dans une table.

Pour retrouver un mot de passe il faut d'abord retrouver à quelle chaîne appartient une empreinte. Pour ce faire on génère une chaîne à partir de l'empreinte donnée jusqu'à ce qu'on trouve un mot de passe qui est égal à une fin de chaîne stockée dans la table. A ce moment on sait que l'empreinte appartient à la chaîne dont le début et la fin ont été stockés – ou à une chaîne qui a une fin identique. En regénérant la chaîne à partir du début qui a aussi été stocké dans la table on retrouve le mot de passe que l'on recherche ou on constate qu'il s'agit d'une *fausse alarme* due au fait que deux chaînes ont la même fin. En effet il existe une chance que des chaînes *fusionnent*, ce qui se produit quand la fonction de réduction génère le même mot de passe à partir de deux empreintes différentes. Ceci est inévitable quand l'ensemble des empreintes possibles est plus grand que l'ensemble des mots de passe considérés.

Le procédé est illustré à la figure 1. A partir de l'empreinte "h8" on génère une chaîne jusqu'à ce que tombe sur le mot de passe "1" qui est une fin de chaîne qui nous avons stocké. On regénère la chaîne à l'aide du début de chaîne qui a aussi été stocké pour trouver le mot de passe "8".

2.1 Tables multiples et arcs-en-ciel

Nous avons vu que dans certains cas deux chaînes peuvent fusionner. Ceci est un problème quand on veut générer de grandes tables. En effet, si beaucoup de chaînes on déjà été stockées dans une table, la probabilité est grande qu'une chaîne supplémentaire fusionne avec une chaîne qui est déjà dans la table. Dans ce cas une partie de la chaîne est donc faite de mots de passe qui sont déjà dans la table. Il vaut mieux dans ce cas recommencer une nouvelle table avec une autre fonction de réduction. Le compromis temps-mémoire consiste à générer ℓ tables de m chaînes de longueur t a l'aide de ℓ fonctions de réduction et de ne stocker que les début et les fins de chaînes.

Une variante plus efficace que nous avons développée récemment consiste à générer des chaînes dans lesquelles on utilise une fonction de réduction différente

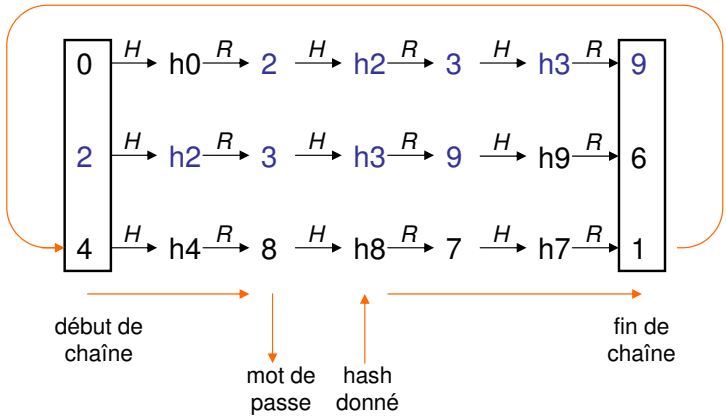

Fig. 1. Recherche du mot de passe "8" à partir de son empreinte "h8" à l'aide de trois chaînes dont on a stocké que le début et la fin.

pour chaque position dans la chaîne. On calcule donc la première réduction avec la fonction R_1, la deuxième avec la fonction R_2 et ainsi de suite. On réduit considérablement la probabilité de fusion, car pour que deux chaînes fusionnent il faut non seulement que le même mot de passe apparaisse dans les deux chaînes, mais qu'en plus il apparaisse à la même position dans la chaîne. Cet artifice permet de générer des tables beaucoup plus grandes ce qui à l'avantage de réduire le nombre d'opérations nécessaires à rechercher un mot de passe. Ce type de table est appelé table *rainbow*, ou arc-en-ciel, parce que chaque chaîne contient tout le spectre des fonctions de réduction. Pour rechercher à partir d'une empreinte un mot de passe dans une table rainbow on suppose d'abord qu'il se trouve dans la dernière colonne. On applique la dernière fonction de réduction et on cherche le résultat dans les fins de chaîne. Si on ne le trouve pas, on suppose que le mot de passe se trouve à l'avant-dernière position des chaînes. On applique l'avant-dernière réduction, un hashage et la dernière réduction pour à nouveau chercher le résultat dans les fins de chaînes stockés. Dans le pire des cas il faudra $\frac{t(t-1)}{2}$ opération de hashage pour trouver un mot de passe dans la table. Comme les table rainbow peuvent être t fois plus grandes que les tables classiques, on gagne au moins un facteur 2 à l'aide des tables rainbow. En réalité ce gain peut être bien plus élevé, les détails de cette méthode sont donnés dans [3].

3 Performances

Pour exprimer les performances d'un compromis temps-mémoire on utilise les variables suivantes :

– T : le temps nécessaires à retrouver un mot de passe, exprimé en nombre d'opérations de hashage ;

- M : la mémoire nécessaire à stocker les tables, mesurée en nombre de chaînes stockées ;
- N : le nombre de mots de passe possibles ;
- P : la probabilité que l'on trouve le mot de passe dans les tables qui ont été préparée.

Tous les compromis basés sur des chaînes suivent la relation suivante :

$$T \approx \frac{N^2}{M^2}$$

En d'autres mots, le produit du temps de cassage et du carré de la mémoire utilisée est constant. Une chiffre que l'on voit souvent en littérature est que les compromis temps-mémoire permettent de réduire le temps de cassage brut de $T = N$ avec une mémoire nulle à un temps égal à $T = N^{\frac{2}{3}}$ en utilisant une mémoire de $M = N^{\frac{2}{3}}$, ce qui correspond à réduire d'un tiers la longueur du mot de passe ou de la clef à trouver. Comme le temps de cassage augmente avec le carré de la mémoire disponible il est important d'utiliser la mémoire de la manière le plus efficace possible. Un première optimisation consiste à utiliser des tables *parfaites*, c'est à dire, des tables dans lesquelles on élimine les chaînes qui fusionnent. Ensuite il faut aussi choisir un format de stockage des chaînes qui utilise qui permette de stocker un maximum de chaînes dans un nombre d'octets donnés.

3.1 Tables parfaites

Les tables parfaites sont des tables dans lesquelles on retire les chaînes qui fusionnent. Dans le cas des chaînes rainbow cette opération est simple car les chaînes qui fusionnent ont des fins de chaîne identiques. Comme on doit de toute façon trier les chaînes d'après leur fin pour faciliter la recherche dans les tables, l'élimination des fusions se fait sans effort. A chaque fois que l'on retire un chaîne qui fusionne avec un autre on élimine pas seulement des mots de passes qui étaient à double mais aussi le début de la chaîne qui contient des mots de passe uniques. Pour compenser cette perte, il faut donc générer plus de chaînes, obtenir le taux de réussite désiré après élimination des fusions. Le nombre de chaînes sans fusions que l'on obtient après génération d'un nombre de chaînes données est illustré à la figure 2. Le nombre maximum de chaînes que l'on peut générer est égal N, car un peut idéalement utiliser chaque mot de passe comme un point de départ d'une chaîne. Dans notre cas, en générant 80 milliards de chaînes on obtiendra pas plus de 35 millions de chaînes sans fusion et on obtient déjà 30 millions de chaînes sans fusion après avoir généré 210 millions de chaînes.

3.2 Configuration optimale

Pour une mémoire M et un taux de succès P donnés nous allons maintenant essayer de trouver la configuration t, m, ℓ optimale qui va produire le temps de cassage le plus petit. Il se trouve que pour des tables parfaites, cette configuration

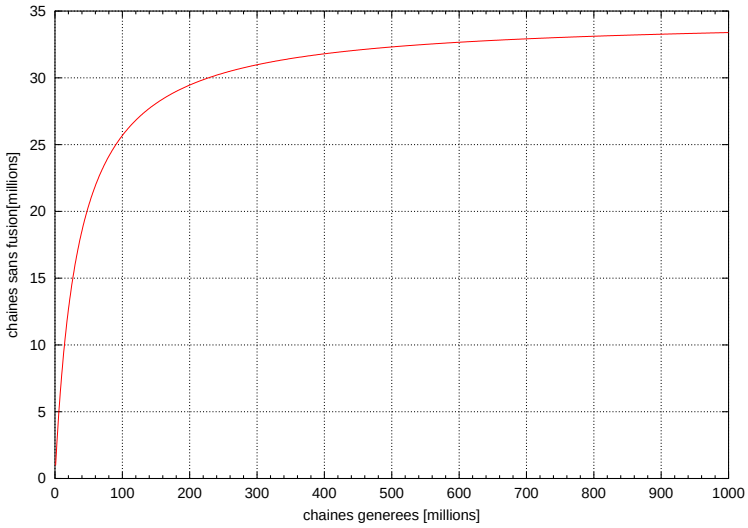

Fig. 2. Effort nécessaire pour obtenir des chaînes sans fusion.

est très simple à calculer. En premier nous allons essayer des tables parfaites aussi grandes que possible, pour limiter le nombre de tables. Le nombre maximal de chaînes de longueur t sans fusions que l'on peut trouver peut être approximé par la formule suivante :

$$m_{max}(t) \approx \frac{2N}{t}$$

Un fait intéressant est que la probabilité de succès d'une telle tables est toujours de 86% indépendamment de N ou t :

$$P_{max} = 1 - \left(1 - \frac{m_{max}}{N}\right)^t \approx 1 - e^{-t\frac{m_{max}}{N}} \approx 1 - e^{-2} = 86\% \qquad (1)$$

La sélection du nombre de tables ℓ est donc simple. Si la probabilité de réussite voulue est plus petite que 86%, il suffit d'une seule table. Sinon, il faut calculer le nombre de tables ℓ qui permet d'arriver au taux de réussite désiré. Comme le nombre de table doit être entier on arrondi vers le haut ce qui a pour effet de baisser le taux de réussite de chaque table un peu en dessous de 86%. Du nombre de tables ℓ découle le nombre de chaînes par table m puisque chaque table aura à disposition une part de mémoire égale à $\frac{M}{\ell}$. Connaissant le nombre de chaînes et le taux de réussite pour chaque table il est facile de calculer la longueur t des chaînes.

$$\ell = \lceil \frac{-\ln(1-P)}{2} \rceil \tag{2}$$

$$m = \frac{M}{\ell} \tag{3}$$

$$t = \frac{\ln(1-P)}{\ln(1-\frac{M}{\ell N})\ell} \approx \frac{N}{M}\ln(1-P) \tag{4}$$

Les formules ci-dessus nous permettent de trouver la configuration optimale du compromis temps-mémoire en fonction de la mémoire disponible et du taux de réussite escompté. Pour pouvoir analyser la performance du compromis il nous reste à calculer combien d'opérations seront nécessaires en moyenne pour retrouver un mot de passe grâce aux tables précalculées.

3.3 Travail à fournir

La structure régulière des tables rainbows permet de calculer exactement le nombre moyens d'opérations nécessaires pour retrouver un mot de passe, en incluant même les opérations dues aux fausses alarmes. Le cassage d'un mot de passe se fait en cherchant dans la dernière colonne de toutes les tables, puis dans l'avant-dernière colonne et ainsi de suite jusqu'à ce que le mot de passe soit trouvé ou que la première colonne est atteinte. Il y a donc au maximum ℓt recherches. Nous calculons le nombre moyen d'opérations de hashage en faisant la somme du nombre d'opérations nécessaires pour trouver un mot de passe dans une recherche donnée pondéré par la probabilité que le mot de passe soit trouvé dans cette recherche.

La probabilité de trouver un mot de passe lors de la k-ème recherche est :

$$p_k = \frac{m}{N}(1-\frac{m}{N})^{k-1}$$

Lors de la k-ème recherche on se trouve dans la colonne

$$c = t - \lfloor \frac{k}{\ell} \rfloor$$

Le nombre d'opérations effectuées pour la k-ème recherche (incluant les recherche précédentes) est égal à

$$T_k = \frac{(t-c)(t-c-1)}{2}$$

La probabilité q_c de tomber sur une fausse alarme lorsque l'on fait une recherche à partir d'une colonne c est égale à la probabilité de tomber sur une fin de chaîne existante moins la probabilité qu'il s'agisse de la bonne chaîne.

$$q_c = 1 - \prod_{i=c}^{i=t}(1-\frac{m_i}{N}) - \frac{m}{N}$$

avec

$$m_t = M \quad \text{et} \quad m_{i-1} = N \ln(1 - \frac{m_i}{N})$$

En combinant ces résultats on trouve que le nombre moyen d'opérations de hashage nécessaires pour retrouver un mot de passe est égal à :

$$T = \sum_{k=0}^{\substack{k=\ell t \\ c=t-\lfloor \frac{i}{t} \rfloor}} p_k \left(\frac{(t-c)(t-c-1)}{2} + \sum_{i=t}^{i=c} q_i i \right) + (1 - \frac{m}{N})^{\ell t} \left(\frac{t(t-1)}{2} + \sum_{i=t}^{i=1} q_i i \right) \quad (5)$$

Le deuxième terme de la somme correspond au travail fourni lorsque le mot de passe n'est trouvé dans d'aucune recherche. Ces formules ont été vérifiées par les mesures suivantes :

$N = 8.06e7, t = 4666,$ $m = 23.8e6, l = 5$	théorie	moyenne de 1000 mesures
calculs d'empreintes (moyenne)	4.30e6	4.11e6
calculs d'empreintes (max)	7.49e7	7.50e7
nombre de fausses alarmes (moyenne)	596	569
nombre de fausses alarmes (max)	12315	12309

Tab. 1. Performances calculées et mesurées des tables rainbow

3.4 Les graphes temps-mémoire

Maintenant que nous savons calculer les performances du compromis dans sa configuration optimale nous pouvons générer les courbes exactes de la relation entre le temps et la mémoire. Dans la figure 3 nous avons représenté ces courbes pour différents taux de réussite.

L'information que l'on retire de la figure 3 est que les tables rainbow génèrent des compromis qui suivent bien la fonction $T \approx N^2/M^2$. Cette relation est valable pour tous les taux de réussite, l'effet du taux est simplement de décaler les courbes. En décrivant l'effet du taux de réussite par une fonction f que nous appelons la *caractéristique* du compromis, nous obtenons un calcul exact du compromis :

$$T = \frac{N^2}{M^2} f(P) \quad (6)$$

La caractéristique du compromis peut être calculée avec les mêmes formules qui ont permis d'établir les courbes temps-mémoire. Le résultat est donné à la figure 4. On y voit comment f évolue en fonction du taux de réussite. Des paliers apparaissent dans la courbe à chaque fois qu'une table supplémentaire

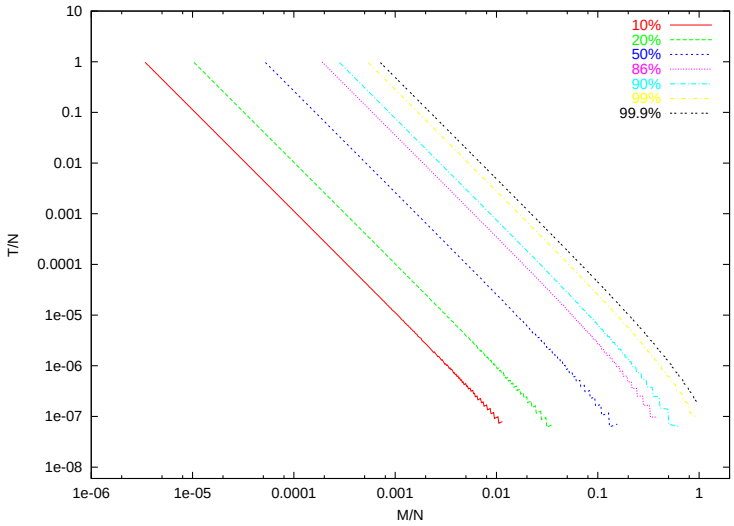

Fig. 3. Courbes temps-mémoire pour différents taux de réussite ($N = 1.6e07$).

est nécessaire pour obtenir le taux de réussite voulu, le premier étant à 86%. Au bas de chaque palier, le taux de réussite par table est proche de 86% par table, ce qui implique un effort énorme pour générer les tables. Il vaut donc souvent mieux "prendre le palier", c'est à dire utiliser une table supplémentaire lorsque le taux de réussite s'approche trop des 86%.

Avec les équations du chapitre 3.2 et le graphe de la caractéristique du compromis, nous avons toutes les informations nécessaires pour trouver la configuration optimale des tables et calculer le temps qu'il faudra pour trouver un mot de passe en fonction de la mémoire disponible. Nous pouvons donc maintenant nous atteler à l'implémentation efficace de notre casseur de mots de passe.

4 Implémentation de l'attaque

Avant de discuter des détails de l'implémentation d'un casseur de mots de passe, nous rappelons brièvement les deux types d'empreintes utilisés par les systèmes Windows.

Le LanManager hash (LM hash) est le type d'empreintes le plus ancien. Il fonctionne avec des mots de passe d'une longueur maximale de 14 caractères. Le mot de passe est d'abord coupé en deux blocs de 7 caractères. Puis toutes les minuscules sont remplacées par des majuscules. Finalement chacun des blocs est utilisé comme clef de 56 bits pour chiffrer un texte prédéfini à l'aide de l'algorithme DES. Les deux résultats de 64 bits chacun sont concaténés pour former le LM hash. Cette manière de faire est très peu efficace. En effet, alors qu'il

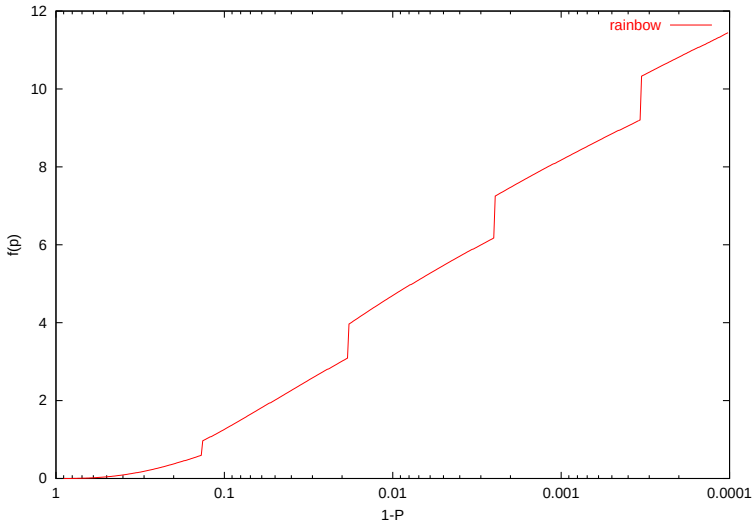

Fig. 4. La caractéristique temps-mémoire des tables rainbow. Les paliers apparaissent à chaque fois qu'une table supplémentaire est nécessaire pour atteindre le taux de réussite désiré.

est possible d'exprimer 2^{84} différents mots de passe alphanumériques de 14 caractères de long, ces mots de passe ne génèreront que 2^{36} moitiés d'empreintes différentes.

Le NT hash est un type d'empreinte plus récent. Il fonctionne avec des mots de passe de longueur arbitraire et ne transforme pas les minuscules en majuscules. L'empreinte est obtenue en appliquant l'algorithme de hashage MD4 sur le mot de passe. Le résultat est une empreinte de 128 bits. Cette méthode génère réellement 2^84 empreintes différentes pour les 2^84 mots de passes alphanumériques possibles.

Pour des raisons de compatibilité il se trouve que toutes les versions de Windows génères les deux types d'empreintes à chaque fois qu'un mot de passe d'un utilisateur est ajouté ou modifié. A partir de Windows 2000 la génération peut être désactivée en modifiant une clef dans le registre Windows. Dans Windows Server 2003 la génération des LM hash est désactivée par défaut.

La méthode la plus efficace pour casser des empreintes Windows est de s'attaquer d'abord aux deux moitiés du LM Hash pour trouver le mot de passe majuscule qui correspond à cette empreinte. Ensuite il ne reste plus qu'a vérifier quelle combinaison de majuscules et minuscules (au plus 16384 possibilités) donne le bon LM hash.

4.1 Optimisations

Les performances réelles d'un casseur à base d'un compromis temps-mémoire dépendent finalement de la vitesse des opération de hashage et du nombre de chaînes que l'on arrive stocker dans la mémoire disponible.

Efficacité du DES : Pour les opérations de hashage il suffit de trouver un bonne implémentation de DES, par exemple celle qui se trouve dans la librairie `libssl`. Notre utilisation de DES a la particularité que la clef de chiffrement change avec chaque nouvelle opération de chiffrement. L'utilisation d'une implémentation de DES en *bitslice* permettrait éventuellement d'augmenter les performances. Ce genre d'implémentation exécute par exemple 32 DES en parallèle en utilisant les mots de 32 bits du processeur pour stocker un bit de 32 différents DES. Les opérations de DES qui se font sur des bits peuvent ainsi être appliquée à 32 calculs en une seule opération de 32 bits. Cette méthode peut même être étendue à 64 bits avec des processeurs de 64 bits ou des opération de 64 bits de processeurs 32 bits (p.ex. MMX sur processeur Pentium). Une implémentation en bitslice est par exemple utilisée par le casseur de mots de passe en force brute `John the Ripper[2]`.

Efficacité du stockage : L'optimisation du stockage des tables est plus important que l'optimisation de DES puisque le temps de cassage diminue avec le carré du nombre de chaînes stockées. Une méthode naïve consisterait à stocker pour chaque chaîne les sept octets qui correspondent au mot de passe du début de la chaîne et les sept octets qui correspondent à la fin. C'est ce qui est par exemple fait dans le logiciel `rainbowcrack`. Au fait, pour faciliter l'alignement des chaînes, ce logiciel utilise 16 octets par chaînes.

Binarisation : La première constatation que l'on doit faire est qu'il n'y a que $2^{36.23}$ mots de passes alphanumériques d'un à sept caractères. On ne devrait donc pas utiliser plus de 37 bits pour stocker un mot de passe. Il suffit de définir une représentation que nous appelons représentation *binaire*, qui assigne à chaque mot de passe un chiffre entre 0 et $2^{36.23}$. Une méthode simple consiste à considérer les mots de passe alphanumériques comme des nombres en base 37 faits des 36 caractères alphanumériques et d'un caractère vide pour les mots de passe courts. En représentation binaire une chaîne peut être stockées à l'aide de 74 bits, ce qui est 1.5 fois plus petit que les 14 octets proposé ci-dessus. Le temps de cassage en est réduit d'un facteur de 2.3 (1.5^2). Au fait, les débuts de chaînes que nous devons stocker sont des mots de passe que nous avons choisis arbitrairement lors de la création des tables. Si nous avons besoins de m_0 débuts de chaînes pour générer les tables, nous choisirons donc les mot de passe qui ont les représentation binaires allant de 0 à $m_0 - 1$. Il nous faudra donc que $\log_2(m_0)$ bits pour stocker les débuts de chaînes. Cette manière de faire permet facilement de stocker un début de chaîne alphanumérique dans un entier de 32 bits.

Indexation des fins de chaînes : Les fins de chaînes peuvent prendre des valeur quelconques parmi les $2^{36.23}$ possibilités. Par contre ces valeurs seront triées par ordre croissant. Les valeurs qui se suivent ont donc des préfixes communs. On peut dès lors ne pas stocker ces préfixes et créer une table d'indexation qui indique à partir de quelle position on trouve un préfixe donné. Un exemple simple consiste à créer 36 fichiers qui correspondent à la première lettre des fins de chaînes et de stocker les chaînes dans les fichiers correspondants. Il n'est donc plus nécessaire de stocker le premier caractère des fins de chaînes puisqu'il est identique à toutes les chaînes stockées dans un fichier. Il se trouve que des mots de passe de 6 caractères alphanumériques peuvent être représentés de manière binaire à l'aide de 32 bits, ce qui permet donc de stocker les fin de chaînes dans un entier de 32 bits. Avec ce qui a été dit précédemment sur les début de chaînes nous pouvons donc stocker une chaîne dans 8 octets, ce qui réduit le temps de cassage d'un facteur 3 par rapport des chaînes de 14 octets ou même un facteur 4 par rapport à des chaînes sur 16 octets comme dans `rainbowcrack`. C'est la solution que nous avons implémentée dans la démonstration que nous avions mis en ligne au cours de l'été 2003[4]. En moins d'un Go nous avons ainsi pu stocker 115 millions de chaînes (5 tables à 23.8 millions de chaînes).

L'utilisation de préfixes plus longs permet d'augmenter encore le gain du à l'indexation. Pour chaque bit supplémentaire que l'on considère dans le préfixe, il y a un bit de moins à stocker pour chaque chaîne mais deux fois plus d'indexes à générer. Dans le cas qui nous intéresse, la solution optimale se trouve autour de 20 bits de préfixe et 16 bits stockés par chaîne. Le stockage de l'index lui-même utilise moins de 10% de la mémoire nécessaire à stocker une table. On arrive donc finalement à 6 octets par chaîne, ce qui permet avec la même quantité de mémoire, de casser les mots de passe 5.4 fois plus vite qu'avec un représentation sur 14 octets.

5 Conclusions

Les compromis temps-mémoire permettent d'obtenir des résultats exceptionnels pour le cassage de mots de passe Windows. Ceci est du à plusieurs raisons. D'abord, les empreintes des systèmes Windows (LM hash et NT hash) peuvent être calculées à l'avance. A notre connaissance, Windows est le seul système d'exploitation actuel ou cela est possible. Dans toutes les variantes d'Unix, par exemple, une valeur aléatoire (le sel) est ajoutée lors de la calculation d'une empreinte. Cette valeur n'étant pas connue à l'avance on ne peut pas préparer des tables à l'avance. La deuxième raison qui rend l'utilisation de compromis temps-mémoire idéale pour casser des mots de passe est qu'il s'agit d'un problème avec une complexité réduite. En effet l'ensemble des mots de passe qui sont potentiellement choisis par des utilisateurs est bien plus petit que les 2^{56} clés possibles pour DES. Si on devait casser un système qui utilise DES avec des clés arbitraires, on arriverait peut-être à casser une clef de 56 bits en quelques heures, mais il faudrait des années de calcul pour préparer les tables !

Outre les mots de passe Windows nous n'avons pas trouvé systèmes qui n'utilisent pas de composante aléatoire (sel, vecteur d'initialisation) et qui sont d'une complexité raisonnablement faible. Nous avons appliqué avec succès notre méthode pour casser le LM hash de mots de passe alphanumériques ($N = 2^{36}$) et des mots de passe faits de chiffres, de lettres et de 16 caractères spéciaux ($N = 2^{40}$). La seule parade proposée par Microsoft contre ce genre d'attaques et de désactiver la génération du LM Hash. Ceci ne peut être qu'une solution momentanée. En optimisant encore un peu l'implémentation et en profitant des performances de machines plus récentes il devient aussi possible de casser directement des NT Hash puisqu'ils ne contiennent pas non plus de sel. Il y a par exemple moins que 2^{48} mots de passes alphanumériques de 8 caractères à casse mixte. Comme les compromis temps-mémoire bénéficient doublement de la loi de Moore (par l'augmentation de la mémoire disponible et de la vitesse des processeurs) il faut espérer que Microsoft ne tarde pas à proposer un nouveau type d'empreintes pour préserver la sécurité des mots de passe de ses systèmes d'exploitation.

Remerciement

Le travail présenté dans cet article a été supporté, en partie, par le Pôle de Recherche National en Systèmes Mobiles d'Information et de Communication (NCCR-MICS), un centre supporté par le Fonds National Suisse pour la Recherche Scientifique (subside 5005-67322).

Références

1. M. E. Hellman, A cryptanalytic time-memory trade off, *IEEE Transactions on Information Theory*, IT-26, pp. 401–406, 1980.
2. John the Ripper, `http://www.openwall.com/john`. Casseur de mots de passe par force brute.
3. Philippe Oechslin, Making a faster cryptanalytic time-memory trade off, *Proceeding of IACR Crypto 2003*, 2003.
4. Philippe Oechslin, Claude Hochreutiner et Luca Wullschleger, Les compromis temps-mémoire ou comment cracker un mot de passe Windows en 5 secondes, *Multisystem and Internet Security Cookbook (MISC)*, numéro 10, 2003.