Etat de l'art sur le cassage de mots de passe

Simon Marechal

Thales Security Systems

1 Introduction

1.1 Les fonctions de hachage

Cet article traite du cassage des mots de passe protégés par des fonctions de hachage cryptographiques. Ces fonctions ont comme principales propriétés de prendre un message de longueur variable en entrée et de créer une sortie de taille fixe. Cette sortie ne doit pas donner d'informations sur l'entrée, que ce soit sa taille, sa structure ou son contenu. A titre d'exemple, voici le résultat de la fonction MD5 pour deux entrées différentes :

```
MD5(test1) = 0x5a105e8b9d40e1329780d62ea2265d8a
MD5(test2) = 0xad0234829205b9033196ba818f7a872b
```

La fonction utilisée transforme deux messages de 5 octets en deux *hachages* de 128bits (16 octets) chacun. De plus, bien que les deux entrées ne diffèrent que de deux bits, les sorties sont totalement différentes. Il n'est donc pas possible en observant simplement la sortie de la fonction MD5 de déduire des informations sur le message en entrée ou de retrouver un message qui aura le même hachage[1].

Une dernière propriété des fonctions de hachage qui seront étudiées ici est qu'elles fonctionnent par *blocs*. C'est à dire qu'elles découpent les messages à hacher en blocs de taille fixe et qu'elles travaillent sur un bloc à la fois. Si la taille du message à hacher n'est pas multiple de la taille du bloc, celui-ci sera complété (c'est l'opération de *padding*) jusqu'à obtenir un bloc complet. A titre d'exemple, la fonction MD5 travaille sur des blocs de 512bits (64 octets).

1.2 Le hachage des mots de passe

Les mots de passe des utilisateurs peuvent être stockés tels quels par les applications. Dans ce cas, lorsqu'un utilisateur tente de s'authentifier, le mot de passe proposé est simplement comparé avec le mot de passe stocké. Ce processus très simple présente un important inconvénient : si il est possible à un des utilisateurs d'accéder à la liste des mots de passe, il pourra se faire passer pour un autre utilisateur sur cette application, ou sur une autre si le même mot de passe est utilisé.

Pour pallier à ce problème, les mots de passe sont le plus souvent stockés sous forme hachée, c'est à dire que seule est stockée la transformée du mot de passe par une fonction de hachage. Un utilisateur malveillant accédant à cette liste ne pourra pas obtenir la liste des mots de passe des autres utilisateurs de manière simple. L'application authentifie les utilisateurs en calculant le hachage des mots de passe saisis et en les comparant avec les hachages stockés.

[1] En pratique ce n'est pas tout à fait vrai car cette fonction a récemment vu certaines de ses propriétés cryptographiques mises en défaut. Mais ces résultats n'ont pas aujourd'hui d'application pour le cassage de mots de passe et seront donc passés sous silence.

Une protection supplémentaire est parfois utilisée pour certains systèmes de hachage des mots de passe. Un composant propre à chaque utilisateur, nommé « sel » (*salt* en anglais, et parfois « grain de sel » en français), est concaténé avec le mot de passe de celui-ci avant d'être haché. Ainsi, même si deux utilisateurs ont le même mot de passe, le hachage sera différent. Ce composant est stocké à côté du hachage. Il peut être calculé aléatoirement, ou simplement être le nom de l'utilisateur.

Pour découvrir le mot de passe en clair qui correspond à un hachage, la seule possibilité est de créer des mots de passe candidats, de les hacher, et de comparer ce hachage avec ceux des mots de passe à casser. En cas de correspondance, on a trouvé un clair valide. En général il est impossible de retrouver un mot de passe bien choisi, mais trivial de trouver un mot de passe faible (comme défini par la suite).

1.3 Historique

Casser les mots de passe est un sport très ancien. On retrouve des messages sur ce sujet sur USENET datant de 1983 [7]. Mais comme on peut le découvrir dans le fameux roman « *The Cuckoo's Egg* »[6], en 1986 un administrateur système lambda n'était pas forcément alerté sur ce sujet. Le programme ayant rendu accessible à tous le cassage de mots de passe est *Crack*. La première version publique date de 1991, et utilisait une version optimisée du système de hachage *Unix Crypt*. Ce programme incluait presque toutes les fonctionnalités d'un système de cassage de mots de passe moderne : règles de mutation des mots de passe candidats, mutualisation du travail sur plusieurs hôtes et optimisation du l'algorithme de hachage.

Par la suite, de nombreux programmes et de nouvelles techniques ont vu le jour. L'une de ces nouvelles techniques est le « *bitslicing* ». En 1997, Eli Biham décrit une nouvelle technique[1] consistant à assimiler par exemple un processeur 64 bits standard à un processeur vectoriel avec 64 vecteurs de 1 bit. En adaptant DES à cette nouvelle représentation, il fut capable de multiplier par 5 la vitesse de crackage des mots de passe Unix sur un processeur standard. Ses résultats ont depuis été améliorés et intégrés à tous les outils sérieux de cassage.

La technique la plus spectaculaire ayant vu le jour est l'utilisation de *Rainbow Tables*[3]. Cette méthode est une amélioration d'une technique existante permettant de réaliser un compromis temps-mémoire très efficace. C'est à dire qu'en précalculant des tables et en stockant les résultats, il est possible de casser de manière très rapide des mots de passe par la suite. La première cible de ce système a été le hachage Windows, le LM hash. Cette méthode est particulièrement efficace contre les mots de passe n'utilisant pas de *sel*. S'il est utilisé, une table par valeur du *sel* doit être calculée, ce qui rend souvent inutilisable cette méthode.

Les programmes

John the Ripper C'est le programme le plus puissant et le plus universel pour qui veut casser des mots de passe. Il est très rapide, stable, capable de casser presque tous les types de mots de passe que l'on peut rencontrer et possède le système de sélection de mots de passe candidats le plus évolué existant. Il rebutera par contre les casseurs amateurs, car il est en ligne de commande, et qu'il est souvent nécessaire de le patcher pour qu'il supporte le cassage de certains formats de mots de passe pourtant très populaires. Une version « pro » non libre est apparue récemment, avec pour objectif de convertir les utilisateurs moins techniques (et plus fortunés) ayant des besoins de cassage de mots de passe.

RainbowCrack Ce programme a démocratisé l'utilisation des *rainbow tables*. Bien que la lecture du code source puisse provoquer des sueurs froides et qu'il ne soit pas vraiment optimisé, il reste le programme de choix de nombre de consultants. Il est très utilisé par des sociétés proposant le cassage de mots de passe contre rémunération.

Ophcrack Ce programme vient du laboratoire du créateur des *rainbow tables*. Il est très bien optimisé et possède une interface graphique le rendant très facile d'accès. Il ne supporte pas autant de types de mots de passe que RainbowCrack. Son format de stockage de tables très efficace a permis la création de « live CD » proposant de démarrer sur ce CD-ROM, d'obtenir automatiquement les mots de passe Windows de l'ordinateur local et de les casser.

CAIN C'est l'outil de choix pour le débutant, en raison de son interface graphique et de sa disponibilité sous Windows. Ce programme fait bien plus que du cassage de mots de passe, et est très complet dans ce domaine. C'est probablement lui qui supporte le plus grand nombre de types de mots de passe à casser. Il est par contre notablement moins efficace que ses concurrents dans ce domaine. De plus, le fait qu'il soit livré uniquement sous une forme binaire protégée par des techniques anti-reverse le rend inutilisable par les plus paranos ou les plus intégristes.

Bob the Butcher Ce programme, censé être l'outil de cassage ultime, en est à sa version beta depuis des années et n'avance pas beaucoup. Il est néanmoins le seul programme moderne permettant un cassage de mots de passe distribué efficace, lorsqu'il fonctionne.

1.4 Les hachages les plus intéressants

Ce chapitre décrit les types de mots de passe les plus souvent attaqués par les consultants en sécurité informatique, et assez probablement par les pirates.

MD5 La vénérable fonction de hachage MD5, bien que mise à mal à plusieurs reprises par des cryptanalystes, est probablement la fonction la plus utilisée pour stocker des mots de passe dans les applications. Les applications Web en sont les premières utilisatrices. Cette fonction présente l'inconvénient d'être très rapidement calculable et d'être souvent utilisée sans *sel*.

LM Hash C'est le format historique dans lequel les mots de passe Windows sont stockés. C'est probablement le système de hachage le plus faible jamais conçu. Un mot de passe subit les opérations suivantes avant d'être haché :

- Transformation de toutes les minuscules en majuscules.
- Découpage du mot de passe en deux parties de 7 caractères.
- Hachage de chacune des deux parties en utilisation une variation de DES.
- Concaténation des deux parties.

En plus de réduire de manière drastique la complexité du mot de passe, cette fonction est facilement « bit-sliçable ».

NT Hash Ce format de stockage des mots de passe est présent depuis très longtemps sur les OS serveurs de Microsoft (d'où son nom), mais il a fallu attendre Windows XP pour qu'il arrive sur le poste utilisateur. Il consiste à hacher le mot de passe (en format unicode) avec la fonction MD4. Il est donc sans *sel*, et rapide à calculer.

MS Cash C'est le format du cache d'authentification du domaine Windows. C'est un cache présent sur les postes clients qui stocke les mots de passe des utilisateurs s'authentifiant sur un domaine. Il est principalement utilisé par les utilisateurs nomades. En effet, sans lui ils ne pourraient pas s'authentifier lorsqu'ils ne sont pas sur le réseau d'entreprise, le contrôleur de domaine étant indisponible. Contrairement aux deux autres formats de hachage Microsoft, celui-ci permet de réaliser une augmentation de privilèges s'il est cassé. Les deux autres formats nécessitent d'être administrateur local pour obtenir les hachages des mots de passer locaux. Celui-ci nécessite d'être administrateur local pour obtenir un mot de passe du domaine, potentiellement un mot de passe d'administrateur du domaine. Il est calculé en hachant en MD4 la concaténation du NT hash et du nom d'utilisateur. Le nom d'utilisateur peut être assimilé à du *sel*, rendant ce format l'un des plus robustes de l'environnement Microsoft.

DES Crypt C'est le format historique utilisés sur les OS de type UNIX pour stocker les mots de passe. Ce format utilise plusieurs fois la fonction DES avec *sel* dont il hérite de deux défauts :

- La taille maximale d'un mot de passe est de 8 caractères sur presque toutes les implémentations, soit la taille d'un bloc DES.
- Le craquage est bit-sliçable.

Bien que ces vulnérabilités aient motivé la création d'une nouvelle génération de fonctions de hachage, cette fonction est plus robuste que les mots de passe précédents par plusieurs ordres de magnitude, principalement en raison de la présence de *sel* et des multiples applications de la fonction DES. Néanmoins, ce *sel* n'a qu'une entropie de 8 bits (256 possibilités) rendant les collisions fréquentes et la création de rainbow tables possible.

BSD MD5 Crypt Cette fonction est aujourd'hui utilisée sur presque tous les Unix open-source, à l'exception d'OpenBSD et de OpenWall Linux. Elle est largement plus robuste que son prédécesseur DES-Crypt car elle n'en a pas les inconvénients. Elle est de plus très lente à calculer, faisant 1000 appels à la fonction MD5, les résultats étant modifiés entre deux appels.

WPA PSK Cette fonction n'est pas une fonction de stockage des mots de passe à proprement parler. Elle est utilisée pour calculer un hachage fonction de divers paramètres (que l'on peut assimiler au *sel*) et du mot de passe partagé entre tous les hôtes présents sur un réseau WIFI protégé par WPA PSK (pre-shared key). Elle est extrêmement lente à calculer, requérant 16396 appels à la fonction SHA1 (en fait, 8198 HMAC-SHA1) et 2 appels à la fonction MD5 (un HMAC-MD5) ainsi que de nombreuses opérations intermédiaires. Le vitesse de cassage de cette fonction est généralement inférieure à une centaine de mots de passe testés par seconde.

Récapitulatif Le tableau suivant donne les vitesses de cassage pour John the Ripper avec des patches récents sur un AMD64 3500+ (cygwin).

Algorithme	Taille maximale (caractères)	Vitesse de cassage brute (K clef / s)	Vitesse avec plusieurs *sels* (K clef / s)
MD5 (Web)	-	5064	-
LM hash (Windows)	7	6628	-
NT hash (Windows)	-	7076	-
MS Cash (Windows)	-	3414	7910
DES crypt (Unix)	8	920	1028
BSD MD5 (Linux)	-	6493	6493

A titre de comparaison, ces chiffres indiquent qu'il faut 672 jours pour tester l'ensemble des mots de passe alphabétiques (lettre minuscules et majuscules) de huit caractères en DES, alors qu'il suffit de 20 minutes avec le LM hash, en raison de l'absence de prise en compte de la casse, et de la limitation de la taille des mots de passe.

2 Optimisation de la vitesse de cassage

Pour améliorer la qualité d'un système de cassage de mots de passe, il est tentant d'augmenter la quantité de mots de passe cassés par unité de temps. La première étape consiste à choisir une architecture matérielle, puis d'optimiser son système de cassage pour celle-ci.

2.1 Processeur standard

Cette architecture est celle utilisée par tous les programmes majeurs de cassage de mots de passe. Elle consiste en un processus fonctionnant sur un unique processeur. Elle permet d'utiliser des méthodes de choix de mot de passe candidat évoluées tout en gardant une structure très simple, permettant une exploitation optimale des ressources d'un processeur.

Réduction du nombre d'opérations Comme pour tout programme, la première étape d'optimisation consiste à réduire la complexité de l'algorithme. Dans le cadre d'une fonction de hachage, les optimisations suivantes sont le plus souvent possibles.

Initialisation de la fonction Les détails varient avec les fonctions de hachage, mais elle consiste le plus souvent à initialiser les variables internes. Il est en général impossible de gagner quoi que ce soit à cette étape.

Préparation du message à hacher Le message doit généralement être complété de sorte que sa taille devienne un multiple de la taille des blocs avec laquelle la fonction de hachage fonctionne. La plupart des fonctions de hachage de ce type travaillent avec des blocs de 64 octets. Les mots

de passe testés ont une taille qui est normalement inférieure à 64 octets, ce qui signifie qu'un seul bloc sera haché. Ce bloc contient en général le message à hacher au début, quelques données fixes en son centre et une valeur fonction de la longueur du message haché vers la fin du bloc. Au lieu d'allouer 64 octets de mémoire, de copier le mot de passe candidat au début, d'ajouter les valeurs fixes et la valeur fonction de la taille du message, puis de libérer la mémoire une fois le hachage connu, il est plus efficace de générer un bloc de 64 octets qui sera utilisé par toutes les opérations de hachage, de générer les mots de passe candidats directement dans ce bloc, et de ne modifier le reste du bloc que lorsque le mot de passe candidat change de taille. Les opérations sur la mémoire sont alors presque toutes éliminées.

Hachage de tous les blocs Comme indiqué précédemment, les fonctions de hachage utilisées ne travaillent en réalité que sur un seul bloc dans le contexte des mots de passe. Il est donc possible de supprimer la partie de l'algorithme qui permet de hacher des messages dont la taille est supérieure à la taille d'un bloc.

Fin du travail Dans cette partie, le hachage est normalement assemblé et peut être comparé avec les hachages des mots de passe à casser. Il est parfois possible d'éviter certains calculs, en particulier les conversions d'endianité finales. Il suffit en effet de les réaliser sur les hachages des mots de passe à casser avant de commencer à tenter de hacher d'autres mots de passe.

Ecriture en assembleur L'écriture en assembleur d'une routine de hachage est le passage obligatoire pour celui qui veut tirer le maximum de son processeur. Les compilateurs actuels, même s'ils produisent du code de très bonne qualité ne sont toujours pas capable d'utiliser correctement les instructions vectorielles, et ne peuvent prévoir certains comportements dus à la gestion de la mémoire cache.

La première étape de l'écriture consiste à convertir l'algorithme en assembleur le plus optimisé possible. Il ne suffit bien sur pas de sélectionner les instructions les plus rapides réalisant l'opération, comme ce peut être le cas sur des processeurs rudimentaires. Sans entrer dans les détails, quatre paramètres sont à prendre en compte par ordre de complexité :

Décodage des instructions C'est là où se fait le gros des optimisations. Les instructions sont classées par ordre de complexité par les fabricants de processeurs. Un processeur moderne est capable d'exécuter plusieurs instructions « simples » en parallèle, alors qu'il ne peut en exécuter qu'une à la fois si elle est complexe. Décomposer une instruction complexe en plusieurs instructions simples est parfois plus efficace. L'exemple le plus connu est l'utilisation du couple « *dec/jnz* » au lieu de l'instruction « *loop* ».

L'autre problématique concerne les dépendances entre les instructions. Ce problème est lié aux optimisations nécessaires pour augmenter les fréquences des processeurs. Les instructions traversent en effet un « pipeline » au long duquel elles sont interprétées. Le problème est que lorsqu'une instruction nécessite des données modifiées par une instruction immédiatement précédente elle est bloquée dans le pipe-line. Le processeur gaspillera alors plusieurs cycles car il fonctionnera à vide, le temps de re-remplir le « pipeline ».

Branchements Lorsqu'un branchement est effectué, c'est à dire lorsque le choix de la prochaine instruction exécutée est fonction d'une condition, le « pipeline » ne peut être rempli par les différentes

zones de codes potentiellement exécutées. Pour pallier à ce problème, la « prédiction de branchement » est utilisée dans les processeurs pour tenter de deviner quelle sera le bon chemin d'exécution et remplir le « pipeline » en conséquence. Lorsque le fonctionnement de l'unité de prédiction est connu, il est possible de l'aider en structurant son programme de sorte à ce qu'elle choisisse la bonne branche le plus souvent possible.

Cache et mémoire Les accès mémoires sont les plus complexes à optimiser. Plusieurs niveaux de mémoire sont disponibles :

- Le stockage (disque dur, mémoire flash)
- La RAM
- Les caches CPU, qui sont de plusieurs types (instruction, données), tailles et vitesses différentes

En général, plus un type de mémoire est rapide, plus il est cher et donc plus sa capacité de stockage est limitée. En ce qui concerne l'optimisation d'un cassage de mots de passe, l'objectif sera de garder autant que possible la boucle de cassage dans le cache de code, et les données à hacher dans le cache de données. Bien sur ces caches sont toujours trop petits, et optimiser leur utilisation (c'est à dire minimiser les transferts entre RAM et cache) devient vite trop contraignant.

Réorganisation des instructions Réorganiser les instructions influe principalement sur le taux de remplissage du pipe-line d'instructions, mais également sur la gestion du cache processeur. Cette méthode d'optimisation consiste, en partant d'un canevas général, à remplacer des blocs d'instructions par d'autres blocs réalisant la même fonction, puis à tester le code résultant pour sélectionner le meilleur candidat. Avec des outils de simulation sophistiqués, tels que des simulateurs de CPU dont la granularité atteint la latence du cache, certaines personnes sont capables de réaliser ces opérations manuellement. C'est surtout le cas dans le monde des supercalculateurs. Pour une personne « normale », la principale approche consiste à écrire un script *Perl* qui va générer de multiples fichiers assembleur, à les compiler et à sélectionner le meilleur. Cette méthode d'optimisation est hautement tributaire de l'architecture étudiée.

Utilisation des fonctions vectorielles Les fonctions vectorielles sont des fonctions permettant de réaliser la même opération sur plusieurs données en parallèle. Ce sont les instructions MMX et SSE du monde intel. Elle sont particulièrement adaptées lorsque de nombreuses opérations élémentaires doivent être effectuées sur une grande quantité de données. Dans le cas du cassage de mots de passe, elles sont surtout utiles avec les fonctions de type MD4, MD5 et SHA1. Sur les processeurs Intel ou AMD, elles permettent de travailler sur 4 entiers de 32bits simultanément, multipliant d'autant, en théorie, la vitesse de calcul. En pratique, les effets de cache et la cruelle absence d'une instruction permettant de réaliser des rotations donne un gain de performance allant de 20% à 200%.

2.2 Processeurs multiples

Le monde du cassage sur processeurs multiples présente deux facettes : plusieurs processeurs sur un même ordinateur, et plusieurs processeurs sur des systèmes différents. Cette différence est de taille, car la quantité bande passante entre les deux processeurs n'est pas comparable. Un processus de cassage de mots de passe est généralement composé de deux composants principaux. L'un d'entre eux génère les mots de passe candidats alors que le second les hache et les compare aux mots de passe à découvrir. Lorsque tous les processeurs sont dans le même système, il est possible d'avoir

un seul composant de génération de mots de passe et un composant de cassage par processeur. Sur les systèmes reliés sur un réseau, il faut qu'un composant de cassage soit situé sur chaque noeud. Cette distinction est importante, et influe sur les stratégies de partage des mots de passe.

2.3 Processeur multicore

Les processeurs multicores peuvent théoriquement permettre de multiplier la vitesse de cassage. Le processeur le plus attrayant est le processeur Cell d'IBM. Il est présent dans la console PlayStation 3 de Sony. Il dispose d'un coeur PowerPC 64 peu efficace et de 7 coeurs spécifiques. Ces coeurs sont des processeurs dédiés au traitement vectoriel, et ont 128 registres généraux de 128 bits. Ils disposent chacun d'une mémoire dédiée très rapide, qui est en pratique un cache de données à gérer manuellement, au moyen de transferts DMA avec la mémoire principale.

Les techniques d'optimisation sont identiques à celles d'un processeur classique, à la différence que les 128 registres permettent en augmentant le parallélisme de garantir le remplissage du pipeline. Les premiers tests ont montré une vitesse de calcul de 7,5M de mots de passe hachés par secondes en MD4 sur un coeur, soit 45M pour 6 coeurs (le septième n'est pas accessible sous linux). Cette vitesse extrême n'est cependant pas exploitable aussi facilement, le coeur PowerPC n'étant probablement pas capable de générer des mots de passe candidats aussi rapidement.

2.4 Les circuits logiques programmables

Les circuits logiques programmables, dont la catégorie la plus connue est le FPGA (field-programmable gate array, réseau de portes programmables in-situ), sont des circuits logiques dont la configuration peut être altérée après leur fabrication. Sans entrer dans des considérations technologiques, ces circuits sont composés de nombreuses unités logiques qui sont librement combinables. Il est possible de créer une puce dédiée de la sorte.

Spécificités de la programmation FPGA La première spécificité est qu'au lieu d'écrire des instructions s'exécutant séquentiellement, il faut décrire le matériel qui sera simulé par le FPGA ! L'approche est complètement différente et permet bien plus de liberté, mais rend nettement plus complexe la conception d'un composant. Plusieurs modules sont décrits : entrées/sorties et fonctionnement. Ces modules sont ensuite assemblés. Après une « compilation[2] » laborieuse, un fichier est créé que l'on peut charger dans sa puce. Des outils de simulation avancés existent, mais sont hors de portée des finances d'un particulier.

En pratique, un module est grossièrement composé de deux parties[3] :

- Une partie « combinatoire », où des signaux sont définis comme fonction d'autres variables.
- Une partie « séquentielle », qui décrit les actions qui sont effectuées à chaque pas d'horloge.

Une fois le comportement décrit, il faut transformer ça en un fichier « programme » pour la puce, ce qui est une opération nettement plus complexe. En effet, une opération qui semble correctement décrite dans le langage HDL choisi peut être synthétisée en quelque chose de tout à fait différent. La plupart du temps, les bugs sont subtils (souvent des *race condition*), et ne se déclenchent que dans des cas particuliers. Et comme il est particulièrement difficile de débugger ces puces, le temps de développement peut être très long.

[2] Le terme est impropre, et cette opération est extrêmement coûteuse. Il faut plus d'une heure pour pouvoir passer du source au fichier de programmation pour un module MD4 simple.

[3] Il n'est pas obligatoire de faire comme ça, mais ça permet d'être proche de la modélisation finale, ce qui est quand même bien pratique pour optimiser ou débugger.

Maximisation de la surface Une fois qu'un coeur fonctionne correctement, il est possible de le dupliquer jusqu'à utiliser toutes les unités logiques de la puce. Ainsi, plus une puce sera « grosse », plus il sera possible d'y insérer des coeurs.

Pipelining Un *pipeline* est un ensemble d'unités de calcul branchées en série, de sorte que la sortie d'une unité soit l'entrée de la suivante. A chaque cycle, toutes les unités effectueront les calculs sur leurs entrées, faisant progresser toutes les informations d'une étape.

La vitesse de calcul globale est égale à la vitesse de calcul de la plus lente des unités. Pour l'accélérer, il est possible de diviser cette unité de calcul en plusieurs unités plus petites, augmentant ainsi la taille du *pipeline*, mais permettant de faire monter la vitesse de l'horloge.

Un système de hachage « pipeliné » prend un mot de passe candidat en entrée et sort un hachage à chaque cycle d'horloge, rendant le système très rapide.

2.5 Comparaison des performances, et perspectives futures

Des tests ont été effectués en implémentant MD4 sur un AMD64 3500+, une PlayStation 3 et un FPGA Spartan XC3S1500. Le code AMD64 est raisonnablement bien optimisé, alors que les codes FPGA et PS3 peuvent très certainement être améliorés[4]. Les résultats ont été les suivants :

Système	Vitesse (Kc/s)	Coût	Kc/s/$
AMD64 3500+	7500	100$	75
XC3S1500	7000	75$	93
PS3	45000	600$	75

Le calcul de la vitesse de cassage par dollar est faussé. En effet, un processeur AMD 3500+ ne sert à rien sans carte mère, carte vidéo, RAM, boîtier, disque dur, ... De la même manière, la PlayStation 3 est un système complet, avec une carte vidéo et un lecteur de Blue-Ray relativement chers. Les FPGA par contre peuvent être installés en très grand nombre sur une même carte, qui peut être créée sur mesure. A titre d'exemple, un constructeur propose une solution contenant 12 XC3S1500 pour 2500$, soit un ratio Kc/s/$ de 33.6.

Le tableau suivant donne le coût en puces FPGA seules pour balayer l'ensemble des mots de passe répondant aux critères indiqués en une journée. Ces taux sont tout à fait approximatifs, mais donnent une bonne idée de l'ordre de grandeur des moyens nécessaires pour casser un mot de passe :

Système	Type	Taille	Coût
LM hash	Alphanumérique	7 (max)	9,7$
LM hash	Alphanum + 13 symboles	7 (max)	94$
NT hash	Alphanumérique	8	27 K$
NT hash	Alphanum + 13 symboles	8	124 K$
NT hash	Alphanumérique	10	104 M$
NT hash	Alphanum + 13 symboles	10	698 M$

Ces résultats montrent notamment qu'un mot de passe de 8 caractères est cassable quelle que soit sa complexité par toute entité, même disposant de faibles moyens financiers. Avec 30.000$ il

[4] Je n'ai en effet passé qu'une semaine sur le FPGA, et une journée sur la PS3, sans aucune connaissance préalable de ces environnements.

est possible de casser n'importe quel mot de passe alphanumérique en une journée, et n'importe quel mot de passe complexe, c'est à dire comprenant des symboles, en une semaine. Les compromis temps-mémoire permettent de réduire encore les moyens nécessaires.

3 Optimisation de la méthode de choix des mots de passe candidats

Un casseur de mot de passe fonctionne donc en hachant des mots de passe candidats et en comparant le résultat avec les mots de passe cible. Si les mots de passe candidats sont bien choisis, c'est à dire si les mots de passe les plus probables sont testés en premier lieu, le cassage de mots de passe sera très efficace. Pour transformer cette constatation en algorithme, le problème est double :

- Comment savoir si un mot de passe candidat a plus de chance de correspondre au hachage qu'un autre ?
- Comment générer efficacement ces mots de passe candidats ?

Une pratique très efficace est l'utilisation d'un dictionnaire de mots de passe communs. Un bon outil de cassage saura lire ce dictionnaire et appliquer des « règles de mutations » (transformation de minuscules en majuscules, ajout de caractères en fin de mot de passe, ...) aux mots de passe individuels. Passé cette étape, un système plus générique doit être utilisé.

3.1 Distribution du travail

La plus importante des difficultés liées au cassage de mots de passe distribué réside dans l'attribution des mots de passe candidats. En effet, la distribution du travail repose sur la capacité qu'à un noeud central de proposer aux noeuds de cassage des espaces de mots de passe à tester. La définition de ces espaces suppose que l'on puisse envoyer à chaque noeud le premier mot de passe qu'il aura à casser. Il doit donc être possible de calculer le nième mot de passe à tester, quel que soit n. De plus, la génération des mots de passe candidats doit être efficace pour les noeuds de cassage, de sorte que cette action ne soit pas pénalisante par rapport au calcul du hachage proprement dit.

3.2 Méthodes statistiques

Méthodes empiriques La méthode « empirique » la plus connue est utilisée par *John the Ripper*. Après avoir analysé un fichier de mots de passe et collecté des informations statistiques, le programme est capable de générer des mots de passe candidats dans un ordre presque optimal[5]. Le principal inconvénient de cette méthode est que bien qu'à partir d'un mot de passe candidat il soit très rapide de trouver le suivant, il est impossible de calculer directement le Nième mot de passe candidat. Cette propriété explique pourquoi les patches MPI et autres variantes distribuées de John n'ont jamais été utilisées pour autre chose qu'afficher des benchmarks avantageux.

Filtres markoviens L'utilisation des filtres markoviens (Du nom de Andrei Markov, le mathématicien russe les ayant inventés) a été appliquée au cassage des mots de passe par Arvind Narayanan et Vitaly Shmatikov dans [2]. Les auteurs appliquent leur technique aux rainbow-tables, mais nous allons voir que ce principe est également très bien adapté au cassage de mots de passe distribué.

Sans entrer dans des considérations mathématiques très clairement décrites dans [2], le concept consiste, pour un filtre de premier degré, à :

[5] C'est bien sûr une affirmation de Solar Designer qui est assez difficile à vérifier.

- Sélectionner un ensemble de mots de passe représentatifs ;
- Calculer les probabilités d'apparition de chaque première lettre x de chaque mot de passe, qui sera par la suite notée $P(x)$;
- Calculer les probabilités d'apparition de chaque lettre x en fonction de la lettre précédente y, notée $P(x|y)$.

La « probabilité markovienne » que nous allons calculer est alors égale au produit de toutes ces probabilités. Par exemple, pour le mot « abcde », $P(abcde) = P(a)*P(b|a)*P(c|b)*P(d|c)*P(e|d)$. Le filtre markovien consiste à ne sélectionner dans un ensemble de mots que ceux dont la probabilité markovienne est supérieure à une certaine valeur. L'énorme avantage de ce système est que, pour un ensemble de mots de passe candidats bien choisi, il est rapide de calculer le Nième mot de passe dont la probabilité markovienne est inférieure à une certaine valeur.

Pour accélérer les calculs, nous ne calculons pas $P(x)$ mais $P'(x) = -log(P(x))$. En effet, si :

$P(abcde) = P(a) * P(b|a) * P(c|b) * P(d|c) * P(e|d)$
alors
$P'(abcde) = P'(a) + P'(b|a) + P'(c|b) + P'(d|c) + P'(e|d)$

Et pour finir, les résultats présentés ici ne sont pas basés pas sur P', mais sur une probabilité corrigée telle que $P_c(x) = P'(x)/longeur(x)$. En effet, à mesure que la longueur d'un mot va augmenter, la probabilité qu'il soit sélectionné va diminuer. Il peut être judicieux de garder $P'(x)$, ou quelque chose comme $P'(x)/sqrt(longueur(x))$ pour explorer plus efficacement l'espace des petits mots de passe.

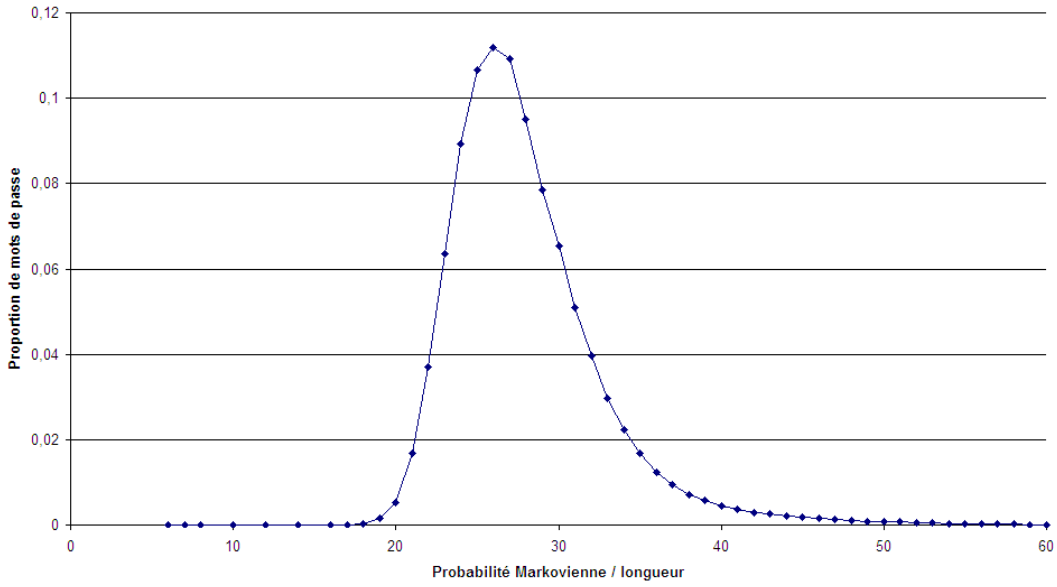

FIG. 1: Distribution de la probabilité Markovienne corrigée

Nous avons réalisé ces calculs sur un dictionnaire de mots de passe important (environ 300.000 mots de passe réels et 100.000 mots provenant de listes publiques). La figure 1 montre la distribution de cette probabilité corrigée dans notre dictionnaire. Nous avons ensuite créé une abaque qui nous permet d'utiliser cet outil. La figure 2 montre la quantité de mots de passe de 10 caractères ou moins qui passent le filtre markovien du premier ordre en fonction de la probabilité cible. C'est à dire la quantité de mots tels que $P_c(mot) < P_{cible} * longueur(mot)$ et $longueur(mot) <= 10$.

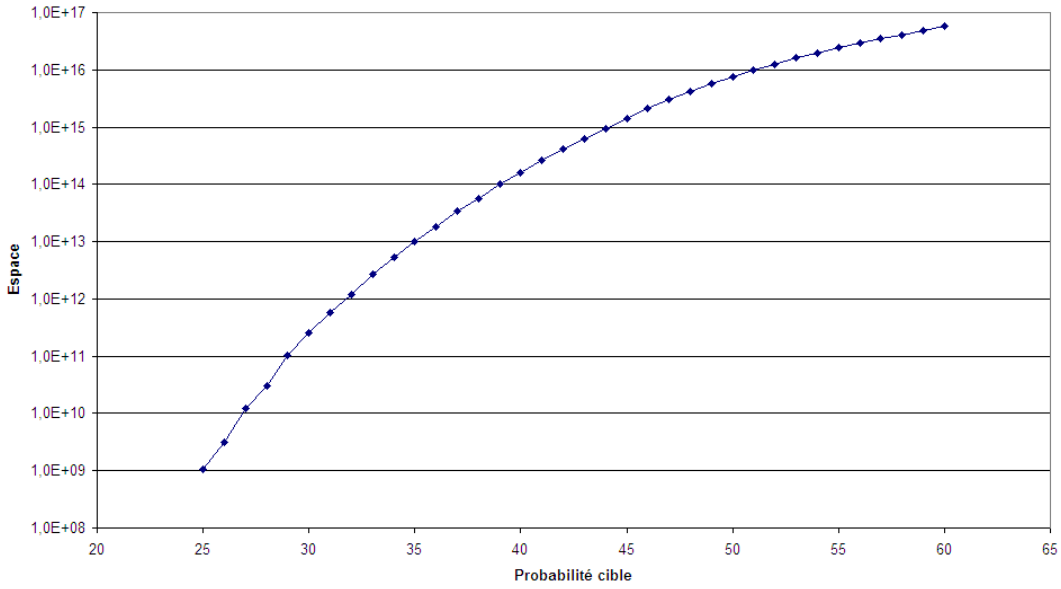

FIG. 2: Espace à parcourir pour les mots de passe de moins de 10 caractères, en fonction de la probabilité cible

Pour utiliser cette abaque, il suffit de :
- Sélectionner les hachages à casser ;
- Calculer la capacité de cassage en mots de passe par secondes du parc de machines utilisé ;
- Diviser cette valeur par la quantité de *sels* différents ;
- Multiplier le résultat par la quantité de temps disponible pour le cassage de mots de passe, pour avoir la quantité de clefs testées durant cette période ;
- Consulter l'abaque pour déterminer la probabilité de markov cible qui correspond à la charge de travail désirée.

Cette méthode n'est aujourd'hui implémentée dans aucun logiciel de cassage de mots de passe.

3.3 Rainbow tables

Une table *arc-en-ciel*, ou *Rainbow Table* est une structure de données permettant de réaliser un compromis temps-mémoire pour le cassage de mots de passe. C'est à dire qu'en précalculant des

tables qui seront stockées (la composant mémoire), il est possible d'accélérer le cassage de mots de passe. Cette méthode, inventée en 2003 par Philippe Oechslin[3] est une amélioration des travaux de Hellman. Pour plus de détails, reportez vous à [4].

Le point important est que ces tables sont composées de chaînes de mots de passe telles que le mot de passe X_n est fonction de X_{n-1}, de la fonction de hachage concernée $H(X)$ et d'une fonction de réduction $R_n(X)$. Pour plus de simplicité, on définit la fonction R_n telle que $R_n(X) = R(X+n)$

$$X_n = R_n(H(X_{n-1})) = R(H(X_{n-1}) + n)$$

Chaînes standards Les rainbow tables, telles qu'implémentées par Rainbowcrack ou Ophcrack, utilisent une « fonction de réduction » qui peut être assimilée à une représentation en base numérique arbitraire. En pratique, si C est le *charset* sélectionné, c'est à dire l'ensemble des caractères pouvant composer le mot de passe, $C(x)$ est le xième caractère de ce *charset* et N la taille du charset, $X_n(x)$ la xième lettre du mot de passe candidat X_n, alors :

$$k = H(X_{n-1}) + n$$

$$X_n(x) = (k/N^x) mod N$$

Cette fonction de réduction est très rapide à calculer, ne nécessitant que deux divisions par caractères. Elle présente l'avantage d'être simple, et de couvrir de manière uniforme l'ensemble des mots de passe candidats si la fonction de hachage est correcte.

Chaînes markoviennes L'utilisation des chaînes de markov dans le contexte des rainbow tables est évoqué par [2]. En utilisant l'abaque présentée précédemment (figure 2), celle présentée figure 3, et les formules « classiques » des rainbow tables, il est possible de calculer la probabilité de découverte d'un mot de passe en fonction des paramètres choisis pour les tables rainbow et pour le filtre markovien.

L'utilisation de cette technique est très intéressante pour les grands mots de passe. Par exemple, il est impossible réaliser un compromis temps mémoire correct avec RainbowCrack sur les mots de passe de 10 caractères ou moins et un *charset* de 64 caractères. Même en choisissant des paramètres disproportionnés (chaînes de 5M maillons, 400M de chaînes par table, 50 tables), le taux de découverte est de 8,18% pour 305Go de tables. En choisissant un taux de compression de 10^9, pour un taux de découverte de 47%, on peut utiliser des paramètres très raisonnables (chaînes de 50K maillons, 400K de chaînes par tables, 1 table) pour une probabilité de découverte de 99% sur l'espace markovien, et donc de 47% sur l'espace des mots de passe et une table de 6Mo.

Ces techniques ne sont aujourd'hui utilisées sur aucun outil.

4 La réalité des mots de passe

Un mot de passe fort est un mot de passe complexe :
- Il doit être long (au moins 10 caractères) ;
- Il doit être composé d'une grande diversité de caractères (au moins une lettre majuscule, une lettre minuscule, un symbole et un chiffre) ;
- Il ne doit pas être dérivé d'un mot connu ;

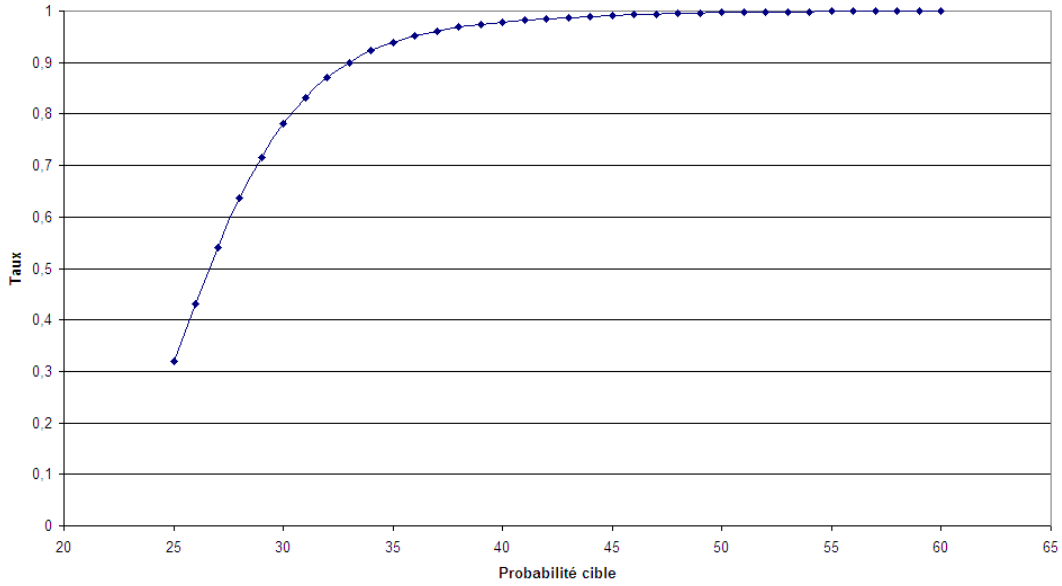

FIG. 3: Taux de mots de passe du dictionnaire couverts en fonction de la probabilité cible

– Si possible, il ne doit pas être prononçable.

Comme on peut le comprendre à la lecture de la liste précédente, un bon mot de passe est un mot de passe dont on ne se souvient pas ! C'est ce qui explique la prolifération des systèmes de gestion des mots de passe. C'est également ce qui explique que si peu de personnes aient des mots de passe forts.

Nous avons observé les faits suivants durant les tests d'intrusion :

– Dans toute grande entreprise il y a au moins un administrateur de domaine Active Directory avec un mot de passe très faible (vide ou mot de passe identique à l'identifiant) ;
– 90% des mots de passe se cassent dans les quinze premières minutes ;
– En fonction du secteur d'activité de l'entreprise, les mots de passe cassés seront majoritairement des prénoms ou des lieux ;
– Il existe presque toujours un « mot de passe d'entreprise » largement répandu ;
– Lorsque la politique de sécurité impose un changement régulier des mots de passe, le nouveau mot de passe est identique à l'ancien à un chiffre près ;
– Il y a presque toujours une base *Notes* ou un fichier *Excel* contenant tous les mots de passe des serveurs. Nous préférons les bases Notes car la fonction de recherche permet de les trouver plus rapidement.

4.1 Recommandations aux utilisateurs

Pour un utilisateur, nous recommandons d'utiliser plusieurs jeux de mots de passe :

- Des mots de passe non critiques, à utiliser sur les applications qui ne sont pas de confiance (applications Web, domaine Windows, accès telnet). Ces mots de passe n'ont pas à être excessivement complexes, car leur hachage réduira de toute façon cette complexité.
- Des mots de passe complexes, très différents les uns des autres, sur les applications de confiance (site bancaire, système Linux). Ces mots de passe, s'ils sont assez complexe, ne seront jamais cassés.

Les systèmes de gestion des mots de passe sont très utiles pour ça, mais seulement si le système sur lequel ils sont stockés est sur ! C'est à dire un système durci, sur lequel l'utilisateur est le seul administrateur.

4.2 Recommandations aux professionels

La première recommandation est d'éviter de réaliser une authentification par mot de passe ! Les solutions de type certificat ou token cryptographique sont obligatoires sur les applications sensibles. Un utilisateur de ces applications est voué à se faire voler son mot de passe, que ce soit à cause de sa faiblesse ou tout simplement des keyloggers.

Sur les applications moins sensibles, il est vital de correctement protéger les hachages des mots de passe, en utilisant un système éprouvé. Pour les applications PHP par exemple, connues pour leur utilisation quasi exclusive du MD5 brut, Solar Designer, le créateur de l'outil John the Ripper, propose une solution[5].

4.3 Petites astuces

Il est possible de créer des mots de passe simples virtuellement incassables. En utilisant par exemple une lettre accentuée, il est possible de tromper tous les craqueurs de mots de passe. En effet, aucun d'entre eux n'a de lettres accentuées dans sa liste de caractères testés par défaut ! Un mot de passe d'une seule lettre peut donc être plus fort contre un craqueur qu'un mot de passe long.

Avoir un mot de passe de plus de 14 caractères permet d'éviter son stockage au format LM. Seul le NT hash, largement plus fort, sera stocké. Mais surtout, tous les pirates / pentesters vont se contenter de tester de casser les LM hash d'un domaine s'ils sont disponibles, laissant de côté les mots de passe uniquement stockés en NT hash.

5 Conclusion

Le cassage de mots de passe est un domaine qui a évolué très rapidement dans les années 90, mais qui stagne aujourd'hui malgré l'invention des rainbow tables. De nouvelles idées sont depuis apparues, mais n'ont été intégrées dans aucun logiciel. L'une des raisons est que les utilisateurs choisissent de toute façon des mots de passe faibles. La raison principale est que le cassage des mots de passe n'est que rarement un enjeu. Un auditeur se contentera d'utiliser un produit qui fonctionne, alors qu'un pirate ne s'intéressera pas aux mots de passe en clair.

Mais le cassage de mots de passe reste le seul domaine de la sécurité des systèmes d'information où un professionnel peut se faire offrir une console de jeu dernier cri par son entreprise.

Références

1. Biham, E. : A Fast New DES Implementation in Software, Technion CSD - CS0981 (1997).
2. Narayanan, A., Shmatikov, V. : Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff, The University of Texas at Austin.
3. Oechslin, P. : Making a Faster Crytanalytical Time-Memory Trade-Off. In Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings, Lecture Notes in Computer Science 2729 Springer 2003, ISBN 3-540-40674-3.
4. Oechslin, P. : `http://lasecwww.epfl.ch/~oechslin/publications/sstic04.pdf`
5. `http://www.openwall.com/phpass`
6. Stoll, C. : *The Cuckoo's Egg*, New York : McGraw-Hill (1994).
7. Usenet : PASSWD CRACKER CONTEST `http://groups.google.com/group/net.general/msg/274445f35a3da963?output=gplain`