Limites des tables Rainbow et comment les dépasser en utilisant des méthodes probabilistes optimisées.

Pierre Lestringant, Philippe Oechslin et Cédric Tissières
pierre.lestringant@epfl.ch
philippe.oechslin@objectif-securite.ch
cedric.tissieres@objectif-securite.ch

Objectif Sécurité SA

Résumé Les tables Rainbow sont un moyen efficace pour casser des hashs de mot de passe non salés. La taille des ensembles de mots de passe pouvant être cassés est limitée par l'effort nécessaire pour créer les tables. L'utilisation de cartes graphiques a permis de repousser un peu cette limite. On trouve ainsi des tables d'une taille de l'ordre de quelques tera octets qui cassent des mots de passe complexes de 8 caractères (10^{16} combinaisons possibles). C'est là qu'apparaît la prochaine limitation : d'aussi grosses tables ne peuvent pas être stockées en RAM et difficilement sur un SSD ce qui rend les temps d'accès très lents et réduit considérablement leur efficacité.

Pour pouvoir passer la limite des 8 caractères, nous avons appliqué deux méthodes probabilistes. La première consiste à représenter les mots de passe par des patterns. Pour les suites de caractères qui apparaissent dans ces patterns, nous utilisons ensuite un modèle de Markov du deuxième ordre pour ne garder que les suites les plus probables. Finalement nous avons optimisé l'implémentation de ce modèle pour l'adapter aux spécificités des GPU.

Nous nous sommes basés sur des bases de données publiques (RockYou, ...) pour choisir les patterns et les pondérations de Markov les plus efficaces et nous avons créé deux jeux de tables : un de taille limitée pour un liveCD Ophcrack et un second de 60 gigaoctets dont les parties les plus importantes peuvent tenir en RAM lors du cassage des mots de passe.

1 Les limites des tables actuelles

1.1 Rappel sur le cassage par table Rainbow

Le but des compromis temps-mémoire est d'accélérer le processus de cassage en permettant d'effectuer un certain nombre d'opérations en amont. Pour ce faire on précalcule les hashs d'un ensemble de mots de passe dont on estime qu'il contiendra les mots de passer à casser. Ces hashs sont stockés de manière condensée sous forme de chaînes. Celles-ci sont composées d'une alternance de mots de passe et de leur hash

associé. Ces chaînes sont calculées à partir de la fonction de hashage et d'une fonction de réduction, qui permet le passage d'un hash à un mot de passe. On note ces fonctions respectivement h_i et r_i, i correspondant, dans le cas des tables Rainbow, à la position dans la chaîne où elles sont utilisées. Les chaînes sont regroupés dans des tables, à raison d'une chaîne par ligne et pour une chaîne donnée d'un couple (mot de passe, hash) par colonne. Cependant pour le stockage des tables, seuls les débuts et les fins de chaînes sont conservés. Pour fixer les idées admettons que les débuts de chaînes soient formés par des mots de passe et les fins de chaînes par des hashs.

Lors de la phase de cassage, on recherche dans chaque colonne de la table. On suppose que le mot de passe est contenu dans le colonne i. On applique les fonctions r_i, h_{i+1} , ... , h_t (où t est le nombre de colonnes) au hash à casser pour retrouver la fin de la chaîne qui lui aurait été associée si celui-ci avait été présent dans la colonne i. On compare alors la fin de chaîne obtenue avec celles contenues dans la table pour vérifier cette hypothèse. En cas d'égalité avec la fin de chaîne de la ligne j, on reconstruit la chaîne j grâce au début de chaîne conservé dans la table. On s'arrête dès que l'on obtient le mot de passe associé au hash à casser situé dans la colonne i. Un exemple est fourni par la figure 1.

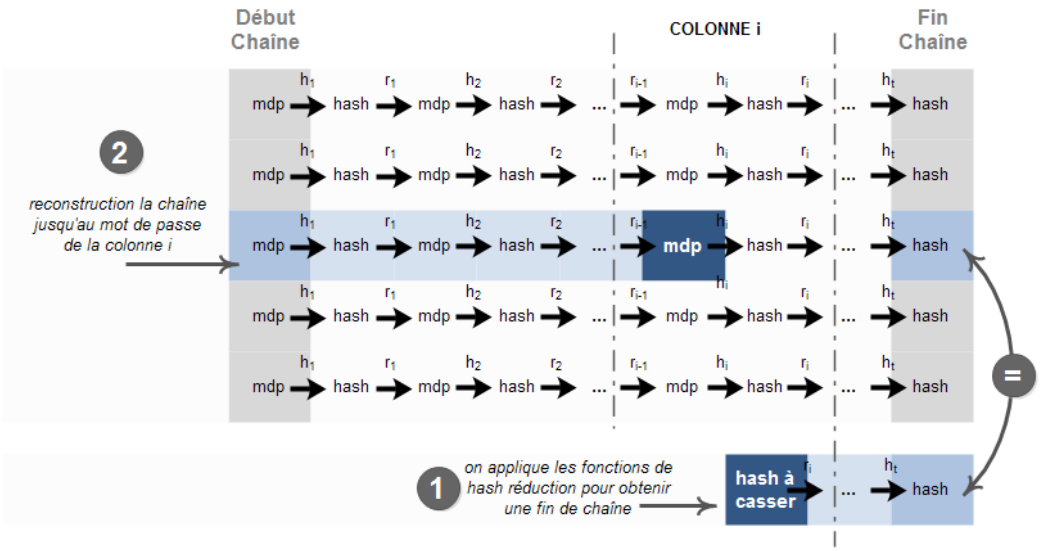

Figure 1. Exemple : lors du cassage à l'aide d'une table précalculée, recherche dans la colonne i

Il est important de remarquer que la fonction de réduction n'est pas injective : l'ensemble de départ pour des hashs de 128 bits est de taille 2^{128} alors que l'ensemble d'arrivée est bien plus restreint, par exemple 2^{53} pour l'ensemble des mots de passe de 8 caractères dans un alphabet d'une centaine de symboles. Pour des hashs distincts la fonction de réduction est donc susceptible de renvoyer des mots de passe identiques. On parle alors de *collision* de la fonction de réduction. Lorsque une telle collision se produit durant la génération d'une table, des chaînes dont les débuts sont pourtant distincts peuvent devenir identiques à partir d'une certaine colonne. On dit alors que ces chaînes fusionnent. L'information contenue par deux chaînes ayant fusionnées est en partie redondante. D'autre part on comprend aisément que trouver une fin de chaîne dans la table n'assure pas que la chaîne associée contienne effectivement le hash recherché. Ce phénomène est appelé faux positif et ralentit le temps de cassage (la vérification des faux positifs peut représenter environ 25% du temps de cassage total).

Les tables Rainbow permettent de limiter ce problème en utilisant des fonctions de réduction différentes pour chaque colonne[8]. Le nombre de fusions est alors fortement réduit : le seul cas de fusion possible est lorsque que la collision de la fonction de réduction apparaît à la même colonne pour les deux chaînes. Les tables Rainbow permettent notamment la création de tables parfaites, dans lesquelles les chaînes ayant fusionnées sont supprimées. Par la suite nous ne considérerons que des tables Rainbow parfaites. Le taux de succès maximal d'une table est autour de 86%[3]. Pour obtenir des taux de succès supérieurs, il est nécessaire d'utiliser plusieurs tables (habituellement autour de quatre).

Pour rechercher un mot de passe dans plusieurs tables deux stratégies sont possibles :

Cassage séquentiel : on recherche le mot de passe dans les tables, l'une après l'autre.A l'intérieur de chaque table on recherche d'abord dans la dernière colonne, puis l'avant-dernière et ainsi de suite. En effet c'est là que l'effort est le plus petit pour trouver une fin de chaîne. Si on ne trouve pas le mot de passe on sera obligé de remonter jusqu'à la première colonne des chaînes puis éventuellement passer à la prochaine table.

Cassage parallèle : on recherche dans toutes les tables simultanément. On va donc chercher dans la dernière colonne de chaque table, puis l'avant-dernière de chaque table et ainsi de suite. L'inconvénient de ce mode de cassage est qu'il nécessite une grande quantité de mémoire : l'ensemble des tables doit être chargé en mémoire pour un fonctionnement optimal

(ce qui n'est, dans le plupart des cas, pas possible étant donné que les tables pouvant faire quelques centaines de Go).

1.2 Limites des tables actuelles

Dans le cas du logiciel Ophcrack, les tables Rainbow classiques (pas d'approche probabiliste) atteignent leur limite pour des mots de huit caractères pris dans un ensemble d'une centaine de caractères (26 lettres majuscules, 26 lettres minuscules, 10 chiffres, et une trentaine de caractères spéciaux). Pour garantir un taux de succès de 99% sur un ensemble aussi vaste (ordre de grandeur du nombre d'éléments : 10^{16}) ces tables demandent 6 mois de calculs sur une machine dédiée, pourvue ici de 3 cartes graphiques AMD Radeon HD6990. Une fois calculées ces tables font 2 To. Le cassage d'un mot de passe prend alors sur un CPU à quatre coeurs en moyenne une dizaine de minutes. En force brute il faudrait générer de l'ordre de 10^{13} hashs à la seconde pour obtenir un résultat équivalent.

Pour comprendre en quoi ces résultats sont de mauvais présages pour l'avenir des tables Rainbow, il faut les mettre en perspective avec les performances atteintes par la force brute. La force brute a bénéficié pleinement de la montée en puissance des GPU de ces dernières années. Pour des NT hashs, la force brute sur un ensemble de 10^{16} éléments demande en moyenne une quinzaine d'heures (pour une configuration identique à celle annoncée pour la génération des tables)[4]. S'il reste intéressant d'utiliser des tables précalculées pour le cassage d'un unique mot de passe, ce n'est en revanche plus vrai pour le cassage de 100 mots de passe (la brute force prend le même temps quel que soit le nombre de mots de passe à casser), sans parler du surcoût demandé par la génération des tables qu'il faudra amortir.

Cependant il faut modérer cette conclusion en gardant à l'esprit que dans le contexte d'une attaque ou d'une analyse forensique, il est plus probable de vouloir casser le mot de passe d'un utilisateur donné plutôt que l'ensemble des mots de passe accessibles. De plus l'attaquant ne dispose pas forcément d'une machine de calcul lui permettant d'atteindre les performances en brute force annoncées, alors que la cassage grâce à des tables précalculées ne demande que peu de puissance de calcul comme nous allons le voir. Enfin dans certains scénarios (*lunch-time attack* par exemple) la contrainte temporelle peut être forte, et même un gain limité (passage de quelques minutes à quelques heures), peut rendre intéressant la conception et le calcul de tables Rainbow.

1.3 Inversion du compromis *temps-mémoire*

La figure 2 montre la diminution du temps de cassage en fonction de la puissance de calcul disponible. Il s'agit de résultats expérimentaux, tirés de l'exécution du logiciel Ophcrack. Deux résultats sont présentés :

- Pour la courbe rouge, l'ensemble des tables est chargé en RAM (ces tables représentent quelques dizaines de Go). La diminution du temps de calcul est proportionelle à l'augmentation du nombre de threads CPU. Le facteur limitant est ici la puissance de calcul.
- Pour la courbe verte, les tables ne sont pas chargées en mémoire et la lecture des fins et des début de chaînes doit se faire systématiquement sur le disque. C'est le scénario dans lequel les tables sont trop volumineuses pour être contenues en RAM ou sur des supports de stockage rapides de type SSD. Dans ce cas, il n'y a pas de diminution du temps de cassage avec l'augmentation de la puissance de calcul disponible. Le facteur limitant est l'accès disque.

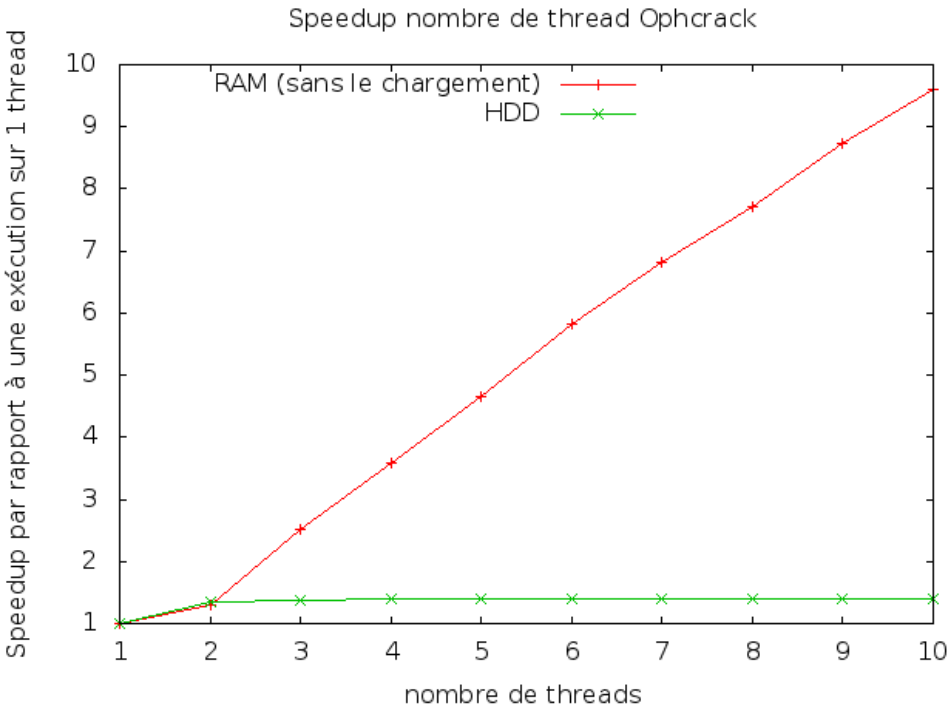

Figure 2. Speedup avec l'augmentation de la puissance de calcul (nombre de threads CPU)

La figure 3 illustre le compromis *temps-mémoire* lorsque l'on prend en compte les accès disque. Il s'agit d'une simulation basée sur la capacité

d'une machine de calcul donnée : 24 cœurs CPU, 48 Go de RAM et 600 Go de SSD. Pour obtenir ces courbes, on fixe la taille de l'ensemble des mots de passe utilisé par les tables (10^{16} éléments environ), ainsi que le taux de succès attendu pour cet ensemble (99.9%). On fait alors varier le nombre de chaînes, pour modifier la taille des tables (attention la relation entre le nombre de chaînes et la taille des tables n'est pas toujours linéaire). La taille des chaînes sert de paramètre d'ajustement pour garantir le taux de succès recherché. On superpose une courbe tendance, proportionelle à $y = 1/x^2$, correspondant au résultat théorique du compromis *temps-mémoire*[8].

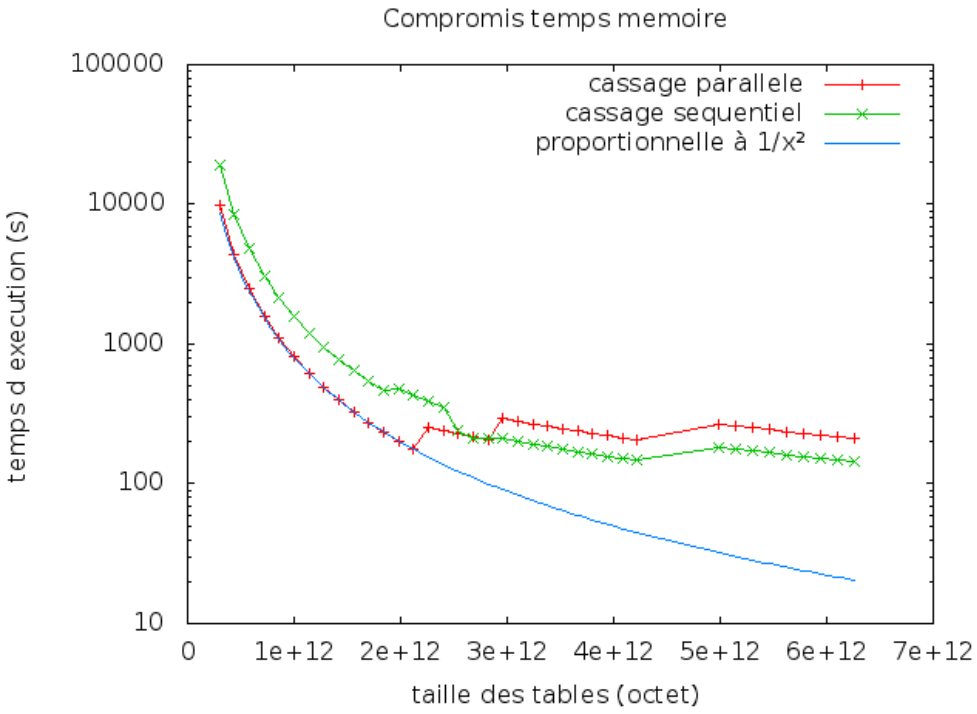

Figure 3. Compromis temps-mémoire : pour 8 caractères, 99.9% de succès

Les décrochages observés sur les courbes sont dus aux déplacements des fichiers de fins de chaînes du SSD vers le disque. Il est intéressant de constater que le cassage séquentiel est plus rapide que le cassage parallèle dans le cas où les accès deviennent pénalisants. Ceci se comprend facilement : le cassage séquentiel requiert davantage de calcul CPU, ce qui ne pose pas problème, tous les threads de calculs n'étant pas occupés et en contrepartie demande moins de mémoire ce qui permet d'utiliser des stockages d'accès plus rapide.

Ceci montre la limite du compromis *temps-mémoire*. Jusque-là les tables étaient suffisamment petites pour que le facteur limitant soit la puissance de calcul. Il était alors intéressant de troquer de la complexité de calcul contre une plus grande consommation mémoire. Aujourd'hui les tables sont volumineuses et les accès disques deviennent le facteur limitant. A partir de là, il n'est plus intéressant de réduire la complexité des calculs. Le compromis inverse se met en place : pourquoi ne pas augmenter la longueur des chaînes pour réduire la taille des tables et ainsi tenter de diminuer les temps d'accès, en utilisant par exemple des SSD ? Cependant le compromis *temps-mémoire* utilisé en sens inverse est très peu efficace : une utilisation à 100% de tous les threads CPU disponibles ne s'accompagnera que d'une faible réduction de la taille des tables. L'utilisation des GPU accentue encore le problème et réduit l'écart qui sépare le cassage par table Rainbow du cassage par brute force.

Dans la partie suivante on s'intéressera aux tables probabilistes. En se limitant aux mots de passe les plus probables, on poursuit un double but : Donner plus de travail aux unités de calcul qui devront faire la différence entre des mots de passes probables et non probables, et réduire la taille des tables nécessaires. Ceci nous permettra de repousser les limites du compromis temps-mémoire et de considérer des mots de passe plus grands.

2 Une approche probabiliste

2.1 Introduction

Pour notre approche probabiliste nous faisons usage de deux méthodes complémentaires. D'une part, nous nous limitons aux mots de passe qui correspondent à des motifs (patterns) fréquents. Ces patterns sont constitués, par exemple, de suites de lettres, de chiffres ou de caractères spéciaux. D'autre part, dans le cas où des suites de lettres apparaissent dans un pattern de mot de passe, nous utilisons un modèle de Markov d'ordre 2 pour sélectionner les suites de lettres les plus probables et ne pas avoir à considérer toutes les combinaisons de lettres possibles.

Une des spécificités des tables Rainbow est que la taille et le contenu de l'ensemble des mots de passe que l'on veut pouvoir retrouver doivent être fixés à l'avance lorsque l'on génère les tables. Ceci crée un problème classique de sac à dos qui consiste à maximiser le taux de succès sans dépasser la taille des tables Rainbow que l'on s'est fixée.

2.2 Définitions

Afin de faciliter la compréhension de la partie qui va suivre on définit un certain nombre d'expressions qui seront réutilisées par la suite.

Ensemble cible : Il s'agit d'un premier sous-ensemble de l'ensemble de **tous** les mots de passe qui a pour fonction de limiter la portée de l'attaque. Il peut être intéressant de restreindre une attaque en fonction du temps disponible, de la capacité de calcul, de la présence ou non d'une *password creation policy*. Il s'agit d'une restriction grossière qui porte sur la longueur des mots de passe, ou sur l'alphabet utilisé.

Ensemble de recherche : L'ensemble de recherche est un sous-ensemble de l'ensemble cible regroupant les mots de passe les plus probables. Certaines des méthodes qui peuvent être utilisées pour former cet ensemble seront détaillées par la suite. Dans le cas d'une attaque classique qui n'utilise pas de méthode probabiliste, l'ensemble de recherche et l'ensemble cible sont identiques.

Ensemble parcouru : Il s'agit de l'ensemble des mots de passe qui sont susceptibles d'être contenus dans les tables Rainbow car ils appartiennent à l'image par la fonction de réduction de l'ensemble des hashs. Pour des facilités d'implémentation, on impose au cardinal de l'ensemble parcouru d'être une puissance de 2. Cependant les méthodes probabilistes utilisées pour former l'ensemble de recherche ne permettent pas de prendre en compte une telle contrainte. Il est donc souvent nécessaire de le tronquer pour former l'ensemble parcouru.

Ensemble capturé : Il s'agit de l'ensemble des mots de passe effectivement contenus dans les tables Rainbow. Cet ensemble diffère de l'ensemble parcouru, car le taux de succès des tables est inférieur à 100%. Capturer 99.9% des mots de l'ensemble parcouru nécessite au moins quatre tables et un temps de génération important : il faut générer beaucoup plus de chaînes que nécessaire pour pallier à la suppression des chaînes ayant fusionné (tables parfaites). Dans certains cas il peut être intéressant de limiter ce taux (par exemple à 95% avec deux tables), ce qui permet avec le temps et la place économisés de considérer des ensembles parcourus plus importants.

Il est capital de distinguer les méthodes permettant de former l'ensemble de recherche et la manière dont celui-ci va être utilisé pour casser des mots de passe. Dans notre cas l'ensemble de recherche va être obtenu par des méthodes probabilistes et il sera exploité par le biais de

tables Rainbow permettant un parcours extrêmement rapide. Cette distinction nous autorise dans le cadre des tables Rainbow à considérer toutes les méthodes habituellement utilisées pour constituer l'ensemble de recherche : analyse fréquentielle des caractères, dictionnaires, règles de déformation, chaînes de Markov, etc. A ce sujet le terme *brute force* est particulièrement ambigu puisqu'il désigne à la fois l'énumération de toutes les combinaisons pour former l'ensemble de recherche et la manière dont il va être parcouru lors de la recherche du mot de passe à casser : comparaison avec tous les éléments.

Ensemble d'apprentissage et de test Les méthodes probabilistes que nous allons utiliser reposent sur des mécanismes d'apprentissage qui font intervenir des ensembles de mots de passe réels. Le choix de ces ensembles est important pour la réussite des méthodes mises en œuvre. Pour la plupart, les ensembles à notre disposition proviennent de la publication de listes suite à la compromission de bases de données sur des sites web. Ces ensembles peuvent être comparés suivant différents critères :

- **Taille :** Plus l'ensemble comporte de mots de passe et plus les statistiques sur cet ensemble seront pertinentes.
- **Langue du site :** Certaines méthodes probabilistes (dictionnaires, chaînes de Markov) fournissent des résultats spécifiques en fonction de la langue.
- **Hash/Clair :** Le problème des listes hashées est que les mots de passe les plus complexes n'ont pas pu être cassés et sont donc absents de celles-ci.
- **Récréatif/professionnel :** On peut faire l'hypothèse que les utilisateurs choisissent des mots de passe plus complexes dans un cadre professionnel que dans un cadre récréatif. Cependant, d'après un sondage, 33% des d'utilisateurs utilisent le même mot de passe pour toutes leurs authentifications sur Internet[9].
- ***Password creation policy* :** Cela augmente la complexité des mots de passe, tout en supprimant les mots de passe les plus simples.

Compte tenu de ces critères, nous avons choisi la liste de mots de passe de RockYou. RockYou est un site qui propose des applications en ligne qui s'interfacent avec différents réseaux sociaux. De ce fait le site conservait les identifiants et mots de passe en clair de ses utilisateurs sur les différents réseaux sociaux concernés. En 2009 une liste de 32 millions de mots de passe est publiée suite au piratage de la base de données par une injection SQL. Le site proposait du contenu essentiellement en langue anglaise. Il s'agit de la plus grande liste de mots de passe en clair

publiée à ce jour et elle est couramment utilisée pour l'évaluation des méthodes probabilistes.

L'évaluation de la pertinence des méthodes probabilistes fait elle aussi intervenir un ensemble de mots de passe réels, l'ensemble de test, qui doit évidemment être différent de celui retenu pour la phase d'apprentissage.

2.3 Utilisation des patterns

Définition Une première approche pour caractériser les mots de passe les plus probables est l'utilisation de patterns. L'ensemble des caractères utilisables est partitionné en sous-ensembles (catégories), quatre dans notre cas : lettres minuscules (l), lettres majuscules (u), chiffres (d) et caractères spéciaux (s). Un pattern est constitué d'une suite de blocs de caractères appartenant à une même catégorie. La taille des blocs fait partie de la définition du pattern. L'ensemble généré par un pattern correspond à l'énumération de toutes les combinaisons de caractères respectant cette décomposition en blocs de catégorie identique. Par exemple le mot de passe **Passw0rd !** appartient à l'ensemble généré par le pattern {u1, l4, d1, l2, s1}.

L'utilisation de patterns n'est pas nouvelle dans le domaine du cassage de mots de passe. On retrouve ainsi cette notion dans le *mask attack* de hashcat[1].

Comparaison avec les règles de déformation On peut considérer les patterns comme une version limitée des règles de déformation de dictionnaires. En effet ces deux notions visent les mêmes mécanismes de création de mots de passe : déformation d'un mot pour le rendre plus difficile à deviner ou pour l'adapter à des *password creation policy* (taille, catégorie des caractères utilisés, ...). Les patterns sont cependant plus restrictifs que les règles de déformation. De nombreuses règles de déformation ne peuvent pas être retranscrites sous forme de patterns : substitution de caractère ('@' à la place de 'a' par exemple), mise au pluriel, duplication de mot, etc. Néanmoins, il est important de remarquer que les ensembles générés par des patterns distincts sont disjoints et que les ensembles générés par les différents patterns forment une partition de l'ensemble des mots de passe. Cette propriété n'est en revanche pas vraie pour les règles de déformation : des mots de passe générés par des règles différentes peuvent être identiques. Ceci rend les règles de déformation difficilement adaptables aux tables précalculées. Certaines règles posent problème : substitution de lettres (dans le cas où le caractère à substituer

est absent, le mot d'origine est alors identique au mot transformé), ajout de caractères (l'ajout d'un caractère peut engendrer un mot déjà présent), opération à une position donnée dans le mot (si le mot est trop court la déformation n'aura pas lieu), etc. Ceci a pour conséquence d'augmenter le nombre de collisions de la fonction de réduction, ce qui diminue le taux de succès des tables. Autre conséquence : il est plus difficile d'extraire les règles de déformation les plus probables d'un ensemble d'apprentissage que d'en extraire les patterns les plus fréquents. En effet l'association mot de passe/pattern est sans ambiguïté alors que dans le cas des règles de déformation un mot de passe peut avoir été engendré par une infinité de règles distinctes ce qui impose l'introduction d'une distance définie de manière empirique afin de restreindre l'analyse aux déformations les plus *simples*.

Les patterns apparaissent comme une alternative aux règles de déformation dans le cas des tables précalculées : s'ils ne permettent pas d'extraire aussi finement les mots de passe les plus probables, ils restent cependant une bonne approximation, facile à mettre en place et qui a l'avantage de ne pas dégrader le taux de collision de la fonction de réduction.

Utilisation dans le contexte de tables précalculées

Choix des patterns Comme mentionné précédemment les patterns peuvent facilement être appris à partir d'un ensemble de mots de passe (ensemble d'apprentissage). Cet apprentissage permet de déterminer les patterns les plus fréquents. Il faut alors sélectionner parmi les patterns rencontrés un sous-ensemble qui formera l'ensemble de recherche, dont la taille se rapproche de celle de l'ensemble parcouru, la taille de celui-ci est fixé en fonction des caractéristiques des tables que l'on souhaite générer. Il s'agit du problème du sac à dos : comment maximiser le taux de succès de l'ensemble de recherche (c'est-à-dire maximiser le cumul des probabilités des patterns sélectionnés) tout en imposant une borne sur le cumul des tailles des patterns sélectionnés. Le problème du sac à dos est non soluble de manière polynomiale ce qui interdit la recherche de la solution exacte (le choix devant s'effectuer parmi une dizaine de milliers de patterns environ).

On choisit de résoudre ce problème grâce à l'heuristique de l'algorithme glouton : on sélectionne les patterns par efficacité décroissante (rapport entre la fréquence dans l'ensemble d'apprentissage et la taille de l'ensemble généré) jusqu'à atteindre la taille limite de l'ensemble

de recherche. On note P_i la probabilité du pattern i dans l'ensemble d'apprentissage et S_i le cardinal de l'ensemble généré par le pattern i. L'efficacité du pattern i s'écrit : $e_i = \frac{P_i}{S_i}$.

Modification de la fonction de réduction La fonction de réduction est légèrement modifiée par rapport à sa version originelle. Classiquement la fonction de réduction associe au hash un index à l'intérieur de l'ensemble parcouru, puis associe de manière bijective un mot de passe à cet index. Dans le cas des patterns, une fois l'index déterminé, il faut rechercher dans quel pattern est situé le mot de passe associé. Cette recherche peut s'effectuer par différentes méthodes (séquentielle, dichotomique, ...) qui seront étudiées plus en détail dans la partie optimisation. On calcule ensuite le sous-index à l'intérieur du pattern qui nous permet de générer classiquement le mot de passe, en tenant compte des restrictions sur les caractères imposées par le pattern.

La fonction de réduction est présentée en pseudo-code ci-dessous. Elle prend en argument un index dans l'ensemble parcouru, et retourne un mot de passe. Les deux étapes sont présentes : recherche du pattern de manière séquentielle et création du mot de passe. Lors de la seconde étape l'alphabet utilisé dépend du pattern et de la position dans le mot de passe : cette dépendance est illustrée par l'appel aux méthodes : *get_alphabet* et *get_alph_size*.

```
reduction_pattern(index){
  size = 0

  for each pattern in list:
    if (size + size(pattern) > index):
      index -= size
      break
    else:
      size += size(pattern)


  for i in range(pwd_length)
    alphabet = get_alphabet(pattern, i)
    alphabet_size = get_alph_size(pattern, i)

    pwd[i] = alphabet[index%alphabet_size]
    index = index/alphabet_size

  return pwd
}
```

Résultats On appelle facteur de réduction le rapport entre le cardinal de l'ensemble cible et le cardinal de l'ensemble de recherche. Le gra-

phique de la figure 4 donne, pour différentes longueurs de mot de passe, le taux de succès en fonction du facteur de réduction. Cette représentation sera réutilisée par la suite afin d'évaluer l'efficacité des approches probabilistes pour extraire les mots de passe les plus probables. Cette courbe est obtenue en faisant varier la taille de l'ensemble de recherche, ceci revenant à faire varier le nombre de patterns sélectionnés à travers l'algorithme du sac à dos. L'apprentissage des patterns et l'évaluation du taux de succès ont été réalisés sur deux sous-ensembles disjoints de RockYou.

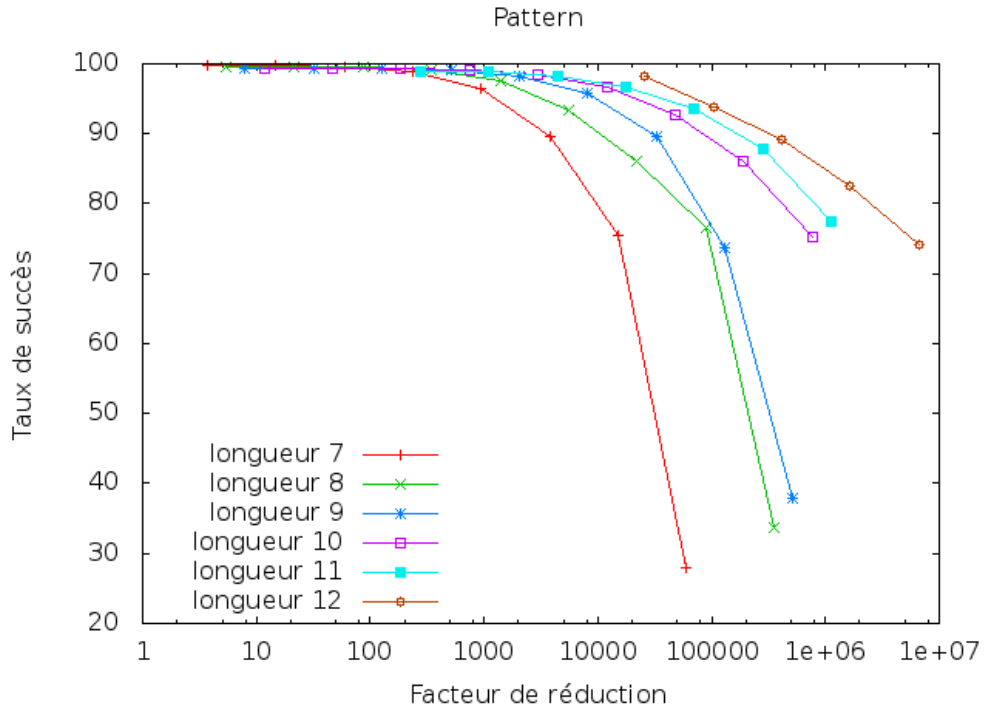

Figure 4. Pattern : taux de succès en fonction du facteur de réduction et de la longueur du mot de passe

L'utilisation des patterns pour limiter la taille de l'ensemble de recherche semble fournir de bons résultats. Par exemple pour des mots de passe de longueur 9, les patterns permettent de générer un ensemble 10 000 fois plus petit que l'ensemble des combinaisons, regroupant néanmoins 95% des mots de passe réels de 9 caractères. On remarque que plus la taille des mots de passe augmente et plus, pour un taux de succès donné, la réduction peut être agressive.

2.4 Utilisation de Markov

Présentation des chaînes de Markov L'utilisation des chaînes de Markov pour restreindre la recherche des mots de passe à des sous-ensembles probables a été introduite en 2005 par Arvind Narayanan et Vitaly Shmatikov [7]. Aujourd'hui il s'agit d'une méthode couramment utilisée dans le domaine du cassage des mots de passe. Elle est basée sur l'hypothèse que les utilisateurs choisissent des mots de passe phonétiquement proches de leur langue pour en faciliter la mémorisation. Cette particularité peut être exploitée par l'utilisation de chaînes de Markov. Il est important de noter que si l'hypothèse laisse espérer de bons résultats pour les chaînes de lettres, elle ne peut pas en revanche s'appliquer aux caractères spéciaux et numériques.

On considère trois ordres pour les chaînes de Markov :

- **Ordre 0 :** la probabilité d'un caractère est indépendante des caractères précédents, ce qui équivaut à une approche fréquentielle. Une amélioration peut être apportée : on utilise une distribution de probabilité différente en fonction de la position du caractère dans le mot (par exemple on peut différencier le premier et le dernier caractère des autres caractères du mot).
- **Ordre 1 :** la probabilité d'un caractère dépend du caractère précédent.
- **Ordre 2 :** la probabilité d'un caractère dépend des deux caractères précédents.

Les probabilités d'occurrence (pour Markov d'ordre 0) et de transition (pour Markov d'ordre 1 et 2) sont mesurées sur un ensemble d'apprentissage. On définit l'improbabilité d'un caractère ou d'une transition comme proportionelle à l'opposé du logarithme de la probabilité. Ceci permet une discrétisation des valeurs qui est indispensable au fonctionnement de l'algorithme d'indexation que nous allons utiliser. On associe à chaque mot de passe une valeur d'improbabilité qui correspond à la somme des improbabilités de chacun de ses caractères. Il devient alors aisé de distinguer les mots de passe les plus probables au sens de Markov, c'est-à-dire ceux dont l'improbabilité est inférieure à un certain seuil. On décide de fixer ce seuil d'improbabilité proportionnellement à la taille des chaînes de Markov que l'on souhaite générer et non d'utiliser un seuil unique quelque soit la taille de la chaîne. Ceci permet de garder un facteur de réduction équivalent quelque soit la taille de la chaîne.

Résultats La figure 5 présente pour les différents ordres de Markov le taux de succès en fonction du facteur de réduction. Comme dans le cas des

patterns, l'apprentissage et l'évaluation du taux de succès sont effectués sur deux sous-ensembles disjoints de RockYou (quelques millions de mots de passe). On se limite dans ce cas aux mots de passe de 9 lettres. Pour tracer ce graphique on effectue différentes mesures en faisant varier le seuil d'improbabilité maximale (une augmentation de ce seuil a pour effet une augmentation du taux de succès et une diminution du facteur de réduction et inversement).

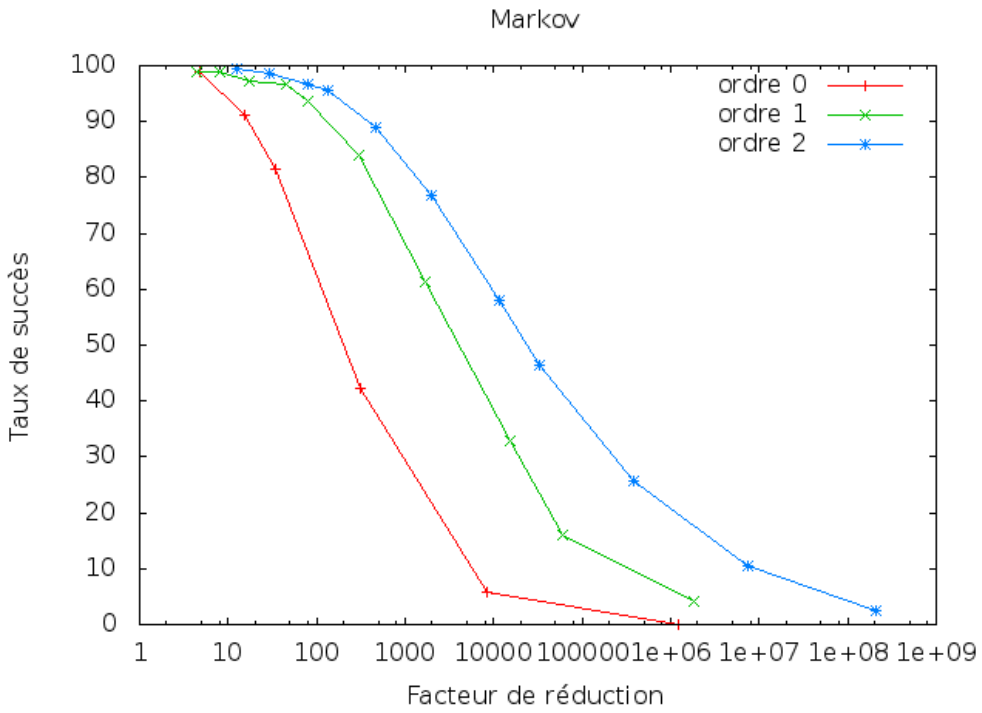

Figure 5. Markov : taux de succès en fonction du facteur de réduction pour des chaînes de Markov de différents ordres

Ces résultats expérimentaux confirment une intuition initiale : plus l'ordre de Markov est important et plus le rapport succès/réduction est intéressant. En effet, si la simple analyse fréquentielle des caractères (Markov d'ordre 0) semble fournir de meilleurs résultats dans certain cas bien particuliers (par exemple les acronymes), elle ne permet pas en revanche de tenir compte de l'organisation des lettres en syllabes à l'intérieur des mots de la langue. Cependant il faut garder à l'esprit que des modèles de Markov d'ordre plus élevés nécessitent davantage de ressources et augmentent le temps d'exécution, comme nous le verrons par la suite. On remarque par ailleurs que le rapport succès/réduction

est bien plus faible que celui obtenu grâce aux patterns : dans le cas d'un modèle de Markov d'ordre 2, pour un taux de succès de 95%, le facteur de réduction est de l'ordre de 100 alors qu'il était de 10 000 dans le cas les patterns.

Utilisation conjointe du modèle de Markov et des patterns Les deux méthodes que nous avons décrites (chaînes de Markov et patterns) sont complémentaires et vont pouvoir être utilisées conjointement. Les patterns apportent une première réduction efficace mais grossière de l'ensemble de recherche. La granularité des patterns ne permet pas d'adapter facilement l'ensemble de recherche à la taille de l'ensemble parcouru par les tables. Les chaînes de Markov vont servir de paramètre d'ajustement, en venant raffiner l'ensemble de recherche. Elles seront utilisées pour réduire la taille des blocs à l'intérieur des patterns. Comme nous l'avons fait remarquer précédemment leur utilisation devra se limiter aux blocs de lettres. Même avec cette restriction les chaînes de Markov restent intéressantes puisque les blocs les plus longs (et donc occupant le plus de place dans l'ensemble de recherche) sont justement les blocs de lettres.

Dans une partie précédente nous avons fait un parallèle entre règles de déformation et patterns. Un second parallèle peut être fait ici, entre les chaînes de Markov et l'utilisation d'un dictionnaire. En effet on comprend que l'on aurait pu utiliser un dictionnaire à la place des chaînes de Markov pour compléter les blocs de lettres à l'intérieur des patterns. C'est cette démarche qu'utilise Matthew Weir dans le cadre de sa grammaire probabiliste [6]. La table 1 fournit pour différents dictionnaires de référence le taux de succès sur l'ensemble RockYou pour les mots de 8 lettres.

Table 1. Taux de succès de différents dictionnaires pour les mots de 8 lettres sur l'ensemble RockYou

Dictionnaire	Nbre mots	Facteur réduction	Succès
Cain & Abel	43511	$4.8\ 10^6$	31.15%
John the Ripper	335	$6.2\ 10^8$	18.41%
500 worst passwords	40	$5.2\ 10^9$	11.19%
62k	87243	$2.4\ 10^6$	56.34%

Le facteur de réduction est extrêmement important par rapport à ce que permettent les chaînes de Markov : pour des mots de huit lettres et un facteur de réduction de l'ordre de 10^6, le taux de succès n'est plus que de 0.5% pour un modèle de Markov d'ordre 2. En revanche le taux

de succès avec les dictionnaires reste faible (il dépasse difficilement les 50%). Pour pallier ce problème, Matthew Weir réintroduit les combinaisons de caractères manquantes mais avec des pénalités pour limiter leur utilisation (*probability smoothing*). Cette démarche a pour but essentiel de pouvoir distinguer à l'intérieur de l'ensemble de recherche les mots de passe les plus probables des mots de passe les moins probables afin de l'ordonner, ce qui est particulièrement intéressant dans le contexte d'une attaque en ligne. Une telle distinction est inutile dans le cas de tables précalculées si bien que l'on préférera l'utilisation des chaînes de Markov. Elles permettent d'engendrer des ensembles bien plus grands que n'importe quel dictionnaire, avec des taux de succès arbitrairement proche de 100%. Ainsi l'utilisation conjointe des patterns et des chaînes de Markov est comparable à l'utilisation d'un dictionnaire avec des règles de déformation.

Modification de l'algorithme du sac à dos L'utilisation de la méthode de Markov pour le pattern i va dégrader son taux de succès. En effet certains mots de passe appartenant initialement à l'ensemble engendré par le pattern, sont composés d'une suite de lettres d'improbabilité supérieure au seuil d'acceptation et ne seront donc plus pris en compte. On note Pm_i la probabilité qu'un mot de passe appartenant au pattern i soit effectivement pris en compte avec l'utilisation de la méthode de Markov sur les blocs de lettres. Le taux de succès global du pattern i avec la méthode de Markov est alors égal à : $P_i Pm_i$. D'autre part la méthode de Markov va permettre de réduire la taille du pattern. On note Sm_i le cardinal de l'ensemble généré par le pattern et dont les éléments satisfont le seuil d'improbabilité de Markov sur les suites de lettres. Plus le seuil d'improbabilité est grand, plus Pm_i est proche de 1 et plus Sm_i est proche de S_i. Si Sm_i peut-être calculé à partir des pondérations de Markov et du seuil d'improbabilité (grâce à l'algorithme de Narayanan et Shmatikov), Pm_i doit en revanche être évalué expérimentalement sur un ensemble d'apprentissage.

Pour un pattern donné il est intéressant d'utiliser la méthode de Markov si l'espace économisé dans l'ensemble de recherche permet de sélectionner d'autre(s) pattern(s) dont le taux de succès compense la perte engendrée par l'utilisation du modèle de Markov : $1 - Pm_i$ (élément initialement présents dans le pattern, mais qui ne satisfont pas le seuil d'improbabilité). Admettons que sans Markov, l'ensemble de recherche permette de sélectionner n patterns (ceux-ci étant classées par

efficacité e décroissante). La condition suivante est nécessaire pour qu'il soit intéressant d'appliquer la méthode de Markov au pattern i :

$$P_i < P_i Pm_i + (S_i - Sm_i)e_{n+1} \quad (1)$$

Le terme de gauche correspond au taux de succès initial (sans l'utilisation de Markov) et le terme de droite est la somme du taux de succès dégradé (du fait du modèle de Markov) et du taux de succès maximal que l'on peut tirer de l'espace économisé (sélection du patern $n+1$). Cette condition n'est pas toujours vérifiée. Le problème de la sélection des patterns se complexifie puisqu'il faut maintenant déterminer conjointement s'il est judicieux d'appliquer un modèle de Markov ou non. Il s'agit d'un cas particulier du problème du sac à dos à choix multiple (0-1 MCKP) pour lequel des heuristiques sont connues. Nous choisissons une méthode itérative basée sur l'inéquation 1 :

1 première sélection sans prendre en compte Markov à l'aide de l'algorithme glouton.
2 on applique l'inéquation pour déterminer sur quels patterns précédemment sélectionnés il faut utiliser la méthode de Markov
3 on met à jour le taux de succès et la taille des patterns pour lesquels la méthode de Markov est utilisée
4 on fait une nouvelle sélection à l'aide de l'algorithme glouton en tenant compte des nouvelles valeurs et on boucle vers l'étape 2

L'algorithme s'arrête lorsque la solution est stabilisée (pour contrer des phénomènes d'oscillation, on peut appliquer le critère à l'étape 2 de manière aléatoire). Théoriquement, nous pourrions laisser à l'algorithme le soin d'optimiser le seuil d'improbabilité pour chaque pattern. Cependant pour des raisons d'implémentation de la méthode de génération des chaînes de Markov, ce n'est pas réalisable et le seuil doit être identique pour tous les patterns. De manière intuitive ce mécanisme de sélection de patterns et du choix du modèle de Markov, va permettre d'associer avec les patterns les plus probables des combinaisons de caractères peu probables (sans Markov) et d'associer à l'inverse avec les patterns peu probables des combinaisons de caractères très probables (chaînes de Markov). Dans le premier cas la dégradation du taux de succès ne vaut pas le gain que pourrait entraîner la sélection de patterns supplémentaires : on ne choisit pas le modèle de Markov et toutes les combinaisons sont considérées. En revanche dans le second cas la dégradation est limitée du fait du faible taux de succès initial du pattern, il est donc intéressant

de choisir le modèle de Markov ce qui revient à ne considérer que les suites de lettres les plus probables. Pour poursuivre dans cette démarche, on pourrait exposer à l'algorithme de sélection un niveau de réduction ultime, réalisé à l'aide d'un dictionnaire par exemple. Mais une telle solution n'a pas été implémentée.

Algorithme de génération Pour la génération des chaînes de Markov on utilise l'algorithme présenté par Narayanan et Shmatikov[7]. Cet algorithme a deux buts : d'une part déterminer la taille de l'ensemble des chaînes de Markov dont l'improbabilité est inférieure au seuil et, d'autre part, à partir d'un index à l'intérieur de cet ensemble, de générer la chaîne correspondante. Il s'agit d'un algorithme de programmation dynamique. Il est expliqué en détail dans l'article d'Alain Schneider[2]. Comme cet algorithme fera l'objet d'une amélioration dans une partie suivante, il est rappelé ci-dessous. Pour des raisons de simplicité il est fourni pour l'ordre 0 de Markov.

La fonction *calculTaille* renvoie le nombre de chaînes de Markov qui obtiennent une valeur d'improbabilité donnée pour une longueur donnée. Pour connaître le nombre de chaînes de Markov inférieur au seuil on appelle la fonction *calculTaille*$(0, 0)$. Elle sera réutilisée par la fonction de génération.

```
calculTaille(longueur, improbaCumul){
  if improbaCumul > seuil:
    return 0
  if longueur == longueur_max:
    return 1
  somme = 0
  for each char in alphabet:
    somme += calculTaille(longueur+1, improbaCumul+improba(char))
  return somme
}
```

La fonction *getChaineMarkov* permet de déterminer la chaîne associée à un index donné, dans le sous-ensemble des chaînes obtenant un score d'improbabilité donné, pour une longueur donnée. Ainsi, pour connaître la chaîne d'index i dans l'ensemble des chaînes les plus probables il faut appeler *getChaineMarkov*$(0, 0, i)$. Cette fonction sera utilisée par la fonction de réduction afin de compléter les blocs à l'intérieur des patterns.

```
getChaineMarkov(longueur, improbaCumul, index){
  if longueur == longueur_max:
    return ""
  for each char in alphabet:
    taille = calculTaille(longueur+1, improbaCumul+improba(char))
```

```
    if taille > index:
      return char | getChaineMarkov(longueur+1, improbaCumul+improba(
          char), index)
    index -= taille
}
```

On remarque que les résultats de la fonction *calculTaille* peuvent être stockés dans un tableau à double entrée : le score d'improbabilité obtenu et la longueur. Une fois ce tableau entièrement calculé lors d'une phase d'initialisation, la fonction *getChaineMarkov* peut alors se contenter de faire un accès mémoire au lieu de recalculer systématiquement la taille des sous-ensembles. Pour des chaînes de Markov d'un ordre supérieur les deux fonctions présentées dépendent du/des caractère(s) précédents. Il est donc nécessaire de multiplier la taille de ce tableau par 26 pour l'ordre 1 et par 676 pour l'ordre 2. Le format de ce tableau peut varier légèrement : dans son article Alain Schneider[2] présente une structure dans laquelle les cases du tableau associent un caractère de l'alphabet à un capital d'improbabilité restant. Cette représentation est particulièrement élégante dans le cas d'une méthode de Markov d'ordre 1 pour la génération de chaînes de taille variable (pas de maîtrise sur la taille de la chaîne retournée). Si elle semble à première vue nécessiter moins d'espace mémoire que la représentation naïve (improbabilité/longueur) décrite dans un premier temps, l'observation de cette dernière montre qu'il s'agit d'une matrice creuse qui peut être condensée lui permettant d'atteindre une taille comparable à la structure proposée par Alain Schneider. Dans une partie suivante, une optimisation est présentée, qui augmente fortement la taille du tableau et impose une nouvelle organisation de celui-ci en mémoire.

2.5 Taux de succès de l'ensemble de recherche

Le graphique de la figure 6 compare le taux de succès de l'ensemble de recherche avec ou sans l'utilisation modèle de Markov (les patterns étant utilisés dans les deux cas). On restreint ici l'ensemble cible aux mots de passe entre 5 et 12 caractères. Le taux de succès est fourni sur l'ensemble cible (il s'agit du pourcentage de mots de passe réels appartenant à la fois à l'ensemble cible et à l'ensemble de recherche). La phase d'apprentissage et d'évaluation du taux de succès ont lieu sur des ensembles disjoints de RockYou.

Pour obtenir un taux de succès de 90%, l'ensemble de recherche doit comporter 2^{47} éléments dans le cas des patterns seuls, contre 2^{42} éléments lorsque l'on associe le modèle de Markov. Ainsi pour un taux de succès

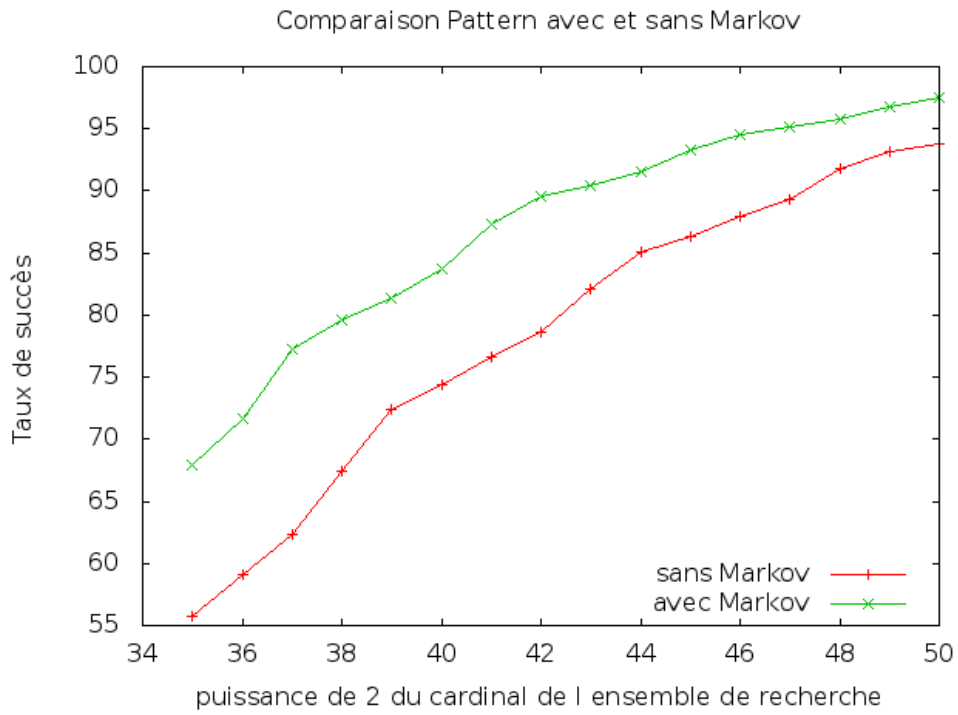

Figure 6. Comparaison du taux de succès de l'ensemble de recherche avec et sans modèle de Markov

équivalent l'ensemble de recherche est 32 fois plus petit avec les chaînes de Markov. Ces résultats expérimentaux sont intéressants car ils permettent de juger de l'intérêt du modèle de Markov en complément des patterns. L'utilisation des chaînes de Markov va augmenter fortement le temps d'exécution de la fonction de réduction : sur une même machine environ 900 millions d'opérations de hash-réduction par seconde pour les patterns seuls, contre 150 millions seulement si l'on rajoute les chaînes de Markov, soit une baisse d'un facteur 6 (des résultats plus détaillés sur le temps d'exécution de la fonction de réduction sont fournis dans la section 3.3). En ce qui concerne le temps nécessaire pour la génération il est donc préférable d'utiliser les chaînes de Markov qui permettent un gain d'un facteur 5 (la fonction de réduction est 6 fois plus lente mais le travail demandé est 32 fois moins important). On aboutit à une conclusion équivalente si l'on compare les temps d'exécution de la phase de cassage. En effet lors de la première partie nous avons vu que compte tenu de la limitation des accès disque, il était tentant d'augmenter la longueur des chaînes afin de réduire la taille des tables. Suivant le compromis *temps mémoire* une diminution d'un facteur 32 de la taille des tables, multiplierait le nombre d'opérations de hash-réduction par 1024. Grâce aux

chaînes de Markov, le temps nécessaire aux opérations de hash-réduction est seulement multiplié par 6 pour une telle diminution.

Le graphique de la figure 7 donne le taux de succès de l'ensemble de recherche sur différentes listes de mots de passe réels. On retrouve parmi ces listes :

- un sous-ensemble de RockYou (disjoint de l'ensemble d'apprentissage)
- la liste Yahoo : 450 000 mots de passe en clair publiés suite au piratage de Yahoo Voice en 2012
- la liste MySpace : 50 000 mots de passe, dont la source serait un pishing sur réseau social MySpace
- *Online liste* : liste de mots de passe provenant du casseur en ligne de Ophcrack

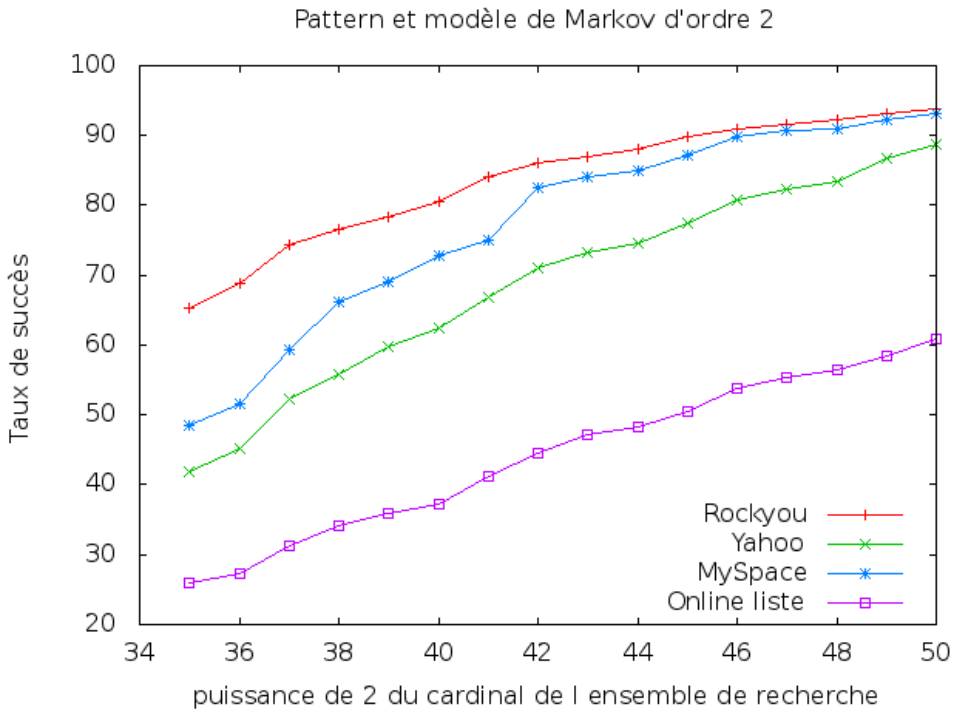

Figure 7. Taux de succès pour différentes tailles d'espace de recherche

3 Optimisation

3.1 Introduction

Pour générer les tables et les utiliser pour casser des mots de passe, nous avons développé plusieurs optimisations qui permettent entre autres d'utiliser au mieux les performances des cartes graphiques. Nous présentons deux de ces optimisations dans les pages suivantes.

La première optimisation est l'utilisation de la méthode *alias* qui permet de générer des nombres aléatoires suivant une distribution biaisée en un nombre fixe et minimal d'opérations. Elle se prête particulièrement bien à l'implémentation sur GPU. Il est intéressant de noter que la méthode obtient son gain en efficacité grâce à l'utilisation de tables précaclulées.

La deuxième optimisation que nous décrivons consiste à utiliser le CPU pour détecter et éliminer des chaînes Rainbow ayant fusionné à mesure qu'elles sont générées par le GPU. Ceci permet typiquement de réduire par deux ou trois la quantité de calculs que doit faire le GPU sans que celui-ci doive s'arrêter pour trier les chaînes ou rechercher des doublons.

3.2 Utilisation de la méthode Alias

Description du problème La méthode de génération des chaînes de Markov (précédemment décrite) fait intervenir une méthode itérative pour déterminer dans quel sous-arbre se situe l'index recherché. Un problème équivalent est aussi rencontré lors de la sélection du pattern au début de la fonction de réduction : il faut déterminer dans quel pattern se situe le mot de passe correspondant à l'index recherché.

Ces problèmes peuvent-être ramenés à la recherche d'un élément dans une liste triée. La complexité est donc $\Theta(n)$ pour une recherche séquentielle et $\Theta(\log(n))$ pour une recherche dichotomique. Des pistes d'amélioration ont été proposées : la recherche séquentielle peut s'effectuer par probabilité décroissante (la recherche commence par les caractères les plus probables ou les patterns générant les ensembles les plus grands), ce qui fait diminuer la complexité moyenne, mais non dans le pire des cas. Ces méthodes sont discutées dans l'article d'Alain Schneider [2]. Cependant ces algorithmes restent particulièrement inadaptés à une implémentation GPU. En effet ils sont composés d'instructions conditionnelles (notamment d'une boucle) et ils comportent un grand nombre d'accès mémoire.

Dans le cas de la génération des chaînes de Markov ces accès mémoires sont particulièrement pénalisants. Compte tenu de la taille du tableau de programmation dynamique utilisé (de l'ordre de la centaine de Mo pour Markov d'ordre 2), il ne peut pas faire l'objet d'une optimisation et être placé dans l'une des zones mémoire spécifiques du GPU (plus rapides ou dotées de stratégies de cache particulières, mais de taille limitée). Il est donc placé dans la mémoire globale qui souffre d'un temps de latence extrêmement important. De plus ces accès sont désordonnés : il n'y a pas de corrélation entre les adresses lues *simultanément* par les différents threads. Ces accès ne pourront donc pas être regroupés (technique habituelle pour accélérer les accès en mémoire globale), et seront exécutés de manière séquentielle.

Nous allons présenter un algorithme qui permet de résoudre ce problème, avec une complexité constante, une seule instruction conditionnelle et un nombre d'accès mémoire réduit.

Présentation de la méthode Alias Plaçons-nous dans le cas des patterns. La fréquence de sélection d'un pattern par la fonction de réduction doit être proportionnelle à la taille de l'ensemble généré par ce pattern pour assurer que tous les éléments de l'ensemble de recherche soient adressés avec la même fréquence par la fonction de réduction. En effet, dans le cas contraire, certains mots de passe produits le plus fréquemment par la fonction de réduction entraîneraient une augmentation du nombre de collisions, et donc un plafonnement plus rapide du taux de succès des tables.

La sélection des patterns peut être assimilée au tirage d'un dé biaisé à n faces : chaque face représentant un pattern avec une probabilité proportionnelle au cardinal de l'ensemble généré rapporté à celui de l'ensemble de recherche. Le problème se résume alors à la modélisation d'un dé biaisé à partir d'un ou plusieurs dés non biaisé (le hash peut être considéré comme un dé non biaisé à 2^{128} faces, ou tronqué pour former plusieurs dés non biaisés).

Lancer de fléchettes Une première approche pour résoudre ce problème est la méthode du lancer de fléchettes. Cette méthode permet de modéliser un dé biaisé à l'aide de deux dés non biaisés. D'un point de vue graphique, les probabilités associées aux différentes faces du dé biaisé sont représentées sous forme d'histogramme. La figure formée est composée de n rectangles dont l'aire est égale à la probabilité de chaque face. L'histogramme est alors utilisé comme cible pour un lancer de fléchettes.

Ce lancer de fléchettes sera réalisé par deux dés non biaisés : un pour les abscisses (ou simplement l'indice de la colonne) et le second pour les ordonnées. Si la fléchette tombe à l'intérieur du rectangle i alors le résultat du dé biaisé vaut i, si la fléchette ne touche aucun rectangle, alors on procède à un nouveau lancer jusqu'à atteindre un rectangle. Cet algorithme est illustré par la figure 8.

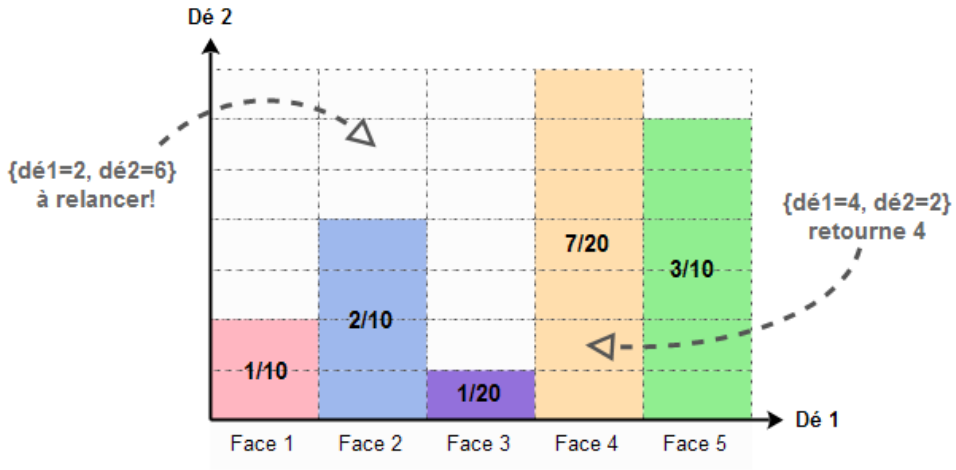

Figure 8. Exemple : modélisation d'un dé biaisé grâce à la méthode du lancer de fléchettes

La complexité de cette méthode est constante dans le meilleur des cas mais infinie dans le pire des cas. En outre cette méthode nécessite de pouvoir relancer les dés en cas d'échec, or on ne dispose que d'un seul hash avec un nombre fixe de bits aléatoires.

Méthode Alias Dans la méthode Alias, les rectangles composant l'histogramme sont découpés afin que leur aire occupe la totalité de la cible. Les rectangles les plus hauts sont tronqués, la partie excédentaire servant à compléter les colonnes les moins remplies. Ce découpage s'effectue selon deux propriétés :

- une colonne est composée de parties provenant d'au plus deux rectangles d'indice différents
- la colonne i débute avec une partie du rectangle i

L'existence d'un tel découpage peut être prouvé [5]. Il se représente facilement sous forme de deux tableaux *Proba* et *Alias*. Pour la colonne i, *Proba*[i] enregistre la taille couverte par le rectangle d'indice i, et *Alias*[i] stocke l'indice du second rectangle (s'il existe) qui vient compléter la

colonne *i*. Comme pour la méthode des fléchettes, on utilise deux dés non biaisés. Le découpage est illustré par la figure 9 qui reprend l'exemple introduit pour le lancer de fléchettes. L'algorithme est décrit ci-dessous :

```
deBiasiseAlias(de1, de2){
  if Proba[de1] > de2:
    return de1
  else:
    return Alias[de1]
}
```

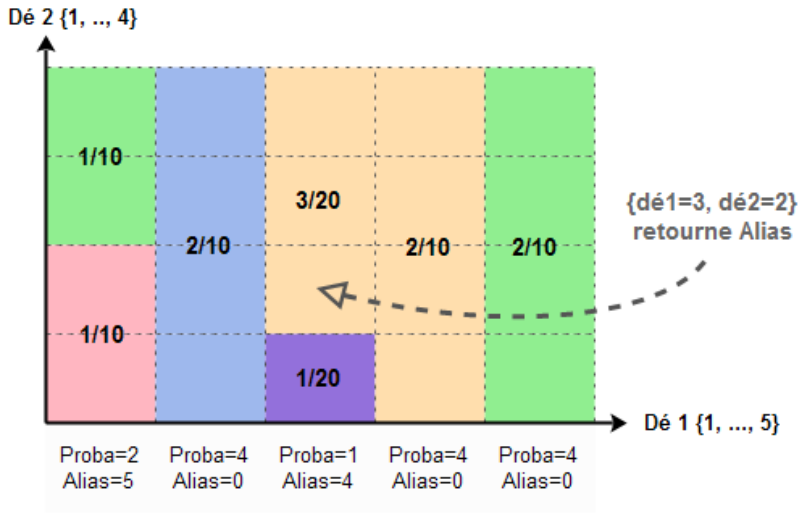

Figure 9. Exemple : découpage utilisé par la méthode Alias

Cette méthode est particulièrement efficace puisque que la complexité dans le pire des cas est constante. D'autre part elle ne fait pas intervenir de boucles (une seule instruction conditionnelle) et se contente de deux accès mémoire. A titre de comparaison, la sélection du caractère dans l'algorithme de génération de Markov, demande au minimum deux accès mémoire par itération (un pour la valeur d'improbabilité et un pour la taille du sous ensemble). Ce qui donne un cas moyen avec 26 accès mémoire pour une recherche séquentielle (13 itérations en moyenne pour parcourir les 26 lettres de l'alphabet).

Initialisation de la méthode Alias Pour l'initialisation de la méthode Alias c'est-à-dire le calcul des tableaux *Proba* et *Alias*, on utilise l'algorithme de Vose [10] qui a une complexité linéaire. Cette étape est réalisée une seule fois par le CPU avant de commencer la génération d'une nouvelle table

ou le cassage d'un mot de passe. Son temps d'exécution est de l'ordre de quelques secondes, c'est-à-dire négligeable par rapport à celui nécessaire aux itérations de la fonction de réduction.

Application à la fonction de réduction La méthode Alias va pouvoir être utilisée à la fois pour les patterns et dans l'algorithme de génération des chaînes de Markov. A titre d'exemple nous fournissons en pseudo code une version de la méthode *getChaineMarkov* (présentée dans la section 2.4) utilisant la méthode Alias. Les arguments sont légèrement modifiés : l'indice est remplacé par deux dés (qui peuvent être assimilés de manière simplifiée aux deux moitiés du hash). Les tableaux *Proba* et *Alias* conservent leur nom, pour permettre de reconnaître l'emplacement de la méthode Alias. Le tableau *Proba* contient la taille du sous ensemble associé à un caractère, une longueur et une valeur d'improbabilité donnée. Le tableau *Alias* contient l'indice d'un caractère de remplacement.

```
getChaineMarkovAlias(longueur, improbaCumul, de1, de2){
  if longueur == longueur_max:
    return ""

  de1_local = de1 % alphabet_size
  de2_local = de2 % calculTaille(longueur, improbaCumul)

  if Proba[de1_local, longueur, improbaCumul] > de2_local:
    char = alphabet[de1_local]
  else:
    char = alphabet[Alias[de1_local, longueur, improbaCumul]

  return char | getChaineMarkovAlias(longueur+1, improbaCumul+improba(
      char), de1/alphabet_size, de2/calculTaille(longueur, improbaCumul)
      )
}
```

Le modèle de Markov va entraîner une augmentation importante de la taille du tableau de programmation dynamique, puisque pour chaque case il est maintenant nécessaire de stocker deux tableaux (*Proba* et *Alias*) de 26 cases chacun. Pour pallier cette augmentation, la structure originelle est conservée et sert à adresser un buffer dans lequel les tableaux *Proba* et *Alias* sont placés les uns à la suite des autres. Le principal avantage de cette méthode est de ne pas avoir à utiliser systématiquement des entiers 64 bits pour le tableau *Proba* mais de pouvoir utiliser des entiers 32 bits lorsque les valeurs mises en jeu le permettent.
Un autre problème est le nombre de bits aléatoires nécessaires. Les différents dés sont détaillés ci-dessous :

– Dé 1 pattern : une face par pattern (≈10 bits)

- Dé 2 pattern : nombre de faces égal au cardinal de l'ensemble d'apprentissage (≈45 bits)
- Dé 1 Markov : 26 faces (≈5 bits).
- Dé 2 Markov : nombre de faces extrêmement variable puisqu'il dépend de la taille du plus grand sous-ensemble considéré. Admettons qu'il demande en moyenne une dizaine de bits aléatoires.

Concernant les dés utilisés pour la génération des chaînes de Markov, il faut un couple de dés pour chaque caractère de la chaîne. Ainsi pour des chaînes de huit caractères il faut environ 175 bits aléatoires au total, ce qui dépasse largement les 128 bits disponibles dans le hash. Une première solution est de réduire la précision des probabilités, ce qui diminue le nombre de faces nécessaires pour le second dé. Une autre solution est de réutiliser plusieurs fois (pour des dés différents) certains bits du hash. Ces deux solutions vont avoir des conséquences négatives sur le nombre de collisions de la fonction de réduction. Cela se traduira par un taux de succès maximal par table plus faible et, pour un taux de succès équivalent, un nombre plus important de chaînes Rainbow à générer. Nous avons choisi d'implémenter la seconde solution, en prenant soin de faire varier les bits réutilisés en fonction de l'indice de la colonne. Les combinaisons impossibles ne le sont que sur une colonne et non sur l'ensemble de la table.

La première itération de chaque chaîne Rainbow devra utiliser une méthode différente pour la fonction de réduction. En effet pour diminuer l'espace de stockage des débuts de chaînes, ceux-ci utilisent le minimum de bits possibles, ce qui interdit l'utilisation de la méthode Alias.

3.3 Suppression des chaînes ayant fusionné

Temps d'exécution de la fonction de hash-réduction Avec l'ajout des algorithmes d'indexation des mots de passe les plus probables (notamment les chaînes de Markov), la fonction de réduction est devenue bien plus lente. Pour donner un ordre de grandeur, elle passe de 1.4 milliard de hash-réductions par seconde pour une implémentation classique à un peu plus de 200 millions de hash-réductions par seconde pour une implémentation probabiliste (temps d'exécution sur une carte AMD Radeon HD6990). Cependant cette dernière valeur varie avec la taille des mots de passe à générer, la fréquence des patterns compressés à l'aide de la méthode de Markov dans l'ensemble de recherche et l'ordre de la méthode de Markov employée. La table 2 donne le nombre d'opérations de hash-réduction par seconde (en millions) dans différents scénarios pour une carte AMD Radeon HD6990.

Table 2. Nombre d'opérations de hash-réduction par seconde selon l'ordre de Markov et la taille maximale des mots de passe sur une carte AMD Radeon HD6990

Taille maximale des mots de passe	Ordre 0 M hashredux/s	Ordre 1 M hashredux/s	Ordre 2 Mhashredux/s
8	311	306	303
9	276	270	266
10	242	231	236
11	222	215	209
12	217	190	146
13	216	188	146
14	199	159	112

La complexité de la fonction de réduction est linéaire par rapport au nombre de caractères. Des ordres de Markov élevés imposent davantage d'instructions conditionnelles et une augmentation du nombre de registres nécessaires, ce qui peut dans certain cas limiter l'occupation du GPU.

La lenteur de la fonction de réduction limite la taille des tables pouvant être générées. Dans le cas d'une machine dédiée, composée de trois AMD Radeon HD6990, la vitesse de génération pour des mots de passe de longueur au plus 10, est aux alentours de 700 millions de hash-réductions par seconde. La table 3 donne un ordre de grandeur du temps de génération pour différentes tailles de tables. Pour simplifier les résultats, on se limite à un cas à quatre tables, avec un taux de succès de 99.8% sur l'ensemble parcouru. On fixe aussi le nombre de colonnes à 40 000 afin de garantir un temps de cassage ne dépassant pas une heure dans le pire des cas. Ces contraintes sont tout à fait cohérentes par rapport à des scénarios de génération réels.

Le temps de génération devient rapidement prohibitif pour des ensembles de taille supérieur à 2^{48}.

Suppression des fusions en arrière-plan Pour diminuer le temps de génération des tables, on va tenter de réduire le nombre d'opérations de réduction nécessaires à leur génération. Dans une approche traditionnelle, les chaînes ayant fusionné ne sont supprimées qu'une fois la génération terminée. Une première amélioration consiste à stopper le programme de génération en cours d'exécution afin de supprimer les fusions. Cependant le temps nécessaire à cette opération s'additionne au temps d'exécution et vient en limiter l'efficacité. L'idée est alors de profiter de la puissance CPU quasiment inutilisée et d'implémenter un mécanisme qui, en arrière-plan, est chargé de supprimer les chaînes ayant fusionné

Table 3. Temps de génération de 4 tables selon la taille de l'ensemble parcouru (Cardinal). Le nombre de chaines Rainbow par table nécessaire pour atteindre le taux de succès de 99.8% est indiqué par m et le nombre de chaînes que l'on doit générer pour obtenir m chaînes après suppression des collisions par m0.

Cardinal	m	m0	Temps
2^{38}	10676408	47835711	3h 2m
2^{40}	42705632	191342844	12h 8m
2^{42}	170822529	765371378	2j 0h
2^{44}	683290116	3061485512	8j 2h
2^{46}	2733160466	12245942051	32j 9h
2^{48}	10932641865	48983768207	129j 14h
2^{50}	43730567462	195935072828	518j 8h
2^{52}	174922269850	783740291313	2073j 9h

au fur et à mesure. Si cette idée ne pose aucun problème conceptuel, son implémentation est en revanche délicate. La suppression des fusions demande, pour une colonne donnée, de trier les fins de chaînes. La complexité de cette opération, $\Theta(n \log(n))$ par rapport au nombre n de chaînes, est donc plus importante que celle de la génération, qui est elle linéaire par rapport au nombre de chaînes. La fréquence maximale à laquelle on va pouvoir supprimer les fusions va donc être limitée pour ne pas impacter le temps d'exécution total (le chemin critique doit toujours être sur le GPU). La valeur de cette fréquence maximale (et optimale) n'est pas facile à estimer de manière théorique et devra être réglée de manière empirique. La formule suivante donne le nombre d'opérations de hash-réduction à effectuer en fonction du nombre de colonnes entre chaque suppression. Rappel sur les notations : m_i est le nombre de chaînes restantes à la colonne i, 2^N correspond au cardinal de l'ensemble parcouru, t est le nombre de colonnes de la table et k est le nombre de suppressions (celles-ci sont réparties régulièrement).

$$Work = \sum_{i=1}^{k} m_i \frac{t}{k} \text{ avec } m_i = \begin{cases} m_0 \text{ pour } i = 0 \\ \frac{m_{i-1} 2^{N+1}}{2^{N+1} - m_{i-1} \frac{t}{k}} \text{ sinon} \end{cases}$$

L'efficacité de la méthode dépend du taux de succès souhaité pour la table : le nombre de fusions croit très fortement avec le taux de succès. Cependant on remarque une tendance générale : le gain converge rapidement avec l'augmentation du nombre de suppressions. Il n'est donc pas intéressant de pousser outre mesure la fréquence des suppressions (quatre ou cinq semblent de bons compromis). Le tableau 4 fournit des estimations du temps de génération avec 4 suppressions. Attention on

fait ici l'hypothèse de ne jamais être limité par la puissance de calcul CPU. En réalité si 4 suppressions semblent expérimentalement tout à fait réalisables pour des ensembles de taille 2^{44} (avec les hypothèses précédentes sur le succès et les colonnes), nous n'avons pas vérifié la validité de l'hypothèse pour des ensembles plus grands.

Table 4. Estimation du temps de génération sans suppression ou avec 4 suppressions selon le cardinal de l'ensemble parcouru

Cardinal	Temps sans suppression	Temps avec suppressions
2^{38}	3h 2m	53m 26s
2^{40}	12h 8m	3h 33m
2^{42}	2j 0h	14h 15m
2^{44}	8j 2h	2j 9h
2^{46}	32j 9h	9j 12h
2^{48}	129j 14h	38j 0h
2^{50}	518j 8h	152j 0h
2^{52}	2073j 9h	608j 0h

4 Exemple de tables

4.1 Tables pour le liveCD Ophcrack

Un premier exemple concret de jeux de tables utilisant les patterns et un modèle de Markov d'ordre 2 est le liveCD d'Ophcrack. On limite ici la taille des tables à 600 Mo au total, pour qu'elles puissent être copiées sur un CD. L'enjeu de ces tables est de considérer l'ensemble de recherche le plus grand possible, tout en respectant la contrainte sur la taille et en garantissant un temps de cassage moyen d'une dizaine de minutes pour un CPU double cœur. On choisit d'utiliser un ensemble de recherche de 2^{39} éléments et de restreindre l'ensemble cible à des mots de passe de 5 à 10 caractères. Afin de réduire le temps de cassage, on décide de réduire légèrement le taux de succès des tables sur l'ensemble parcouru (99.7%) et de n'utiliser que trois tables.

Les résultats obtenus par ce jeu de tables sur les ensembles de mots de passe réels sont présentés dans la table 5. La colonne *Ensemble Cible* indique le pourcentage de mots de passe de la liste appartenant à l'ensemble cible. Ces valeurs permettent d'une part d'évaluer la justesse du choix de l'ensemble cible comme première approximation et d'autre part d'avoir un aperçu sur la complexité relative des différentes listes. La colonne *Succès Cible* fournit le taux de succès pour les mots de passe de

la liste appartenant à l'ensemble cible. Ceci permet de rendre compte des résultats des méthodes probabilistes indépendamment du choix de l'ensemble cible. La colonne *Succès Total* fournit le taux de succès sur la totalité de la liste.

Table 5. Part des mots de passe de la liste appartenant à l'ensemble cible des tables LiveCD et taux de succès de ces tables sur l'ensemble cible ou la totalité de différentes listes de mots de passe

Liste	Ensemble cible	Succès Cible	Succès Total
Rockyou	89.2%	83.9%	74.8%
Yahoo	84.6%	73.5%	62.2%
MySpace	98.4%	93.3%	91.8%
Online liste	68.0%	45.0%	30.6%

4.2 Tables de 60 Go

Un deuxième jeu de tables plus grand a été calculé en utilisant également les patternes et un modèle de Markov d'ordre 2. Les contraintes pour la création de ce jeu étaient de restreindre l'ensemble cible à des mots de passe de 5 à 12 caractères et d'obtenir une taille totale finale des tables inférieure à 60 Go, afin qu'elles puissent être mises facilement sur un SSD et que les index et les fins de chaîne puissent être chargés en RAM. Nous avons opté pour un ensemble de recherche de 2^{46} éléments avec un taux de succès de 99% avec quatre tables.

Les résultats obtenus par ces tables sur les ensembles de mots de passe de test sont décrits dans la table 6.

Table 6. Part des mots de passe de la liste appartenant à l'ensemble cible des tables de 60Go et taux de succès de ces tables sur l'ensemble cible ou la totalité de différentes listes de mots de passe

Liste	Ensemble cible	Succès Cible	Succès Total
Rockyou	95.6%	89.5%	85.6%
Yahoo	97.4%	81.3%	79.2%
MySpace	99.0%	97.8%	96.8%
Online liste	80.4%	62.9%	50.6%

On peut remarquer que les taux de succès sont notablement plus hauts que les tables LiveCD, comme attendu. La différence est particulièrement

intéressante sur des ensembles de mots de passe sur lesquels les tables LiveCD n'obtenaient pas un taux de succès élevé. Cela confirme que des petites tables obtiennent un très bon taux de succès sur des mots de passe très probables, alors que seul un jeu de tables plus importantes permet de casser avec efficacité des mots de passe moins fréquents et plus complexes. Rappelons également que ces tables sont 100 fois plus grandes que celles du LiveCD.

4.3 Comparaison avec les tables classiques

Afin d'évaluer la performance et la pertinence des tables générées, nous les avons comparées à d'autres tables "classiques" d'Ophcrack. Par classique, on entend que la fonction de réduction choisit un mot de passe dans l'ensemble cible de manière équiprobable. Les tables utilisées sont décrites dans la table 7.

Table 7. Liste des tables Ophcrack "classiques" utilisées en comparaison

Nom	Taille	Alphabet	Longueur	Taux de succès
Vista num	3 Go	numérique	1 à 12	99.9%
Vista special XL	107 Go	alphanum. et spéciaux	1 à 7	99%
Vista eight XL	2 To	alphanum. et spéciaux	8	99%

Pour effectuer ces comparaisons, nous avons utilisé les mêmes ensembles de test que pour les sections précédentes. La table 8 permet de comparer les résultats des différentes tables ou combinaisons de tables sur chacun des ensembles. Les tables "Vista proba LiveCD" et "Vista proba 60G" désignent respectivement les tables décrites dans les sections 4.1 et 4.2.

Table 8. Comparaison du taux de succès sur divers ensembles de test en utilisant différentes tables ou combinaisons de tables

Tables utilisées	Rockyou	Yahoo	MySpace	Online
Vista num et Vista special XL	52.4%	30.9%	56.6%	26.5%
Vista num, Vista special XL et Vista eight XL	70.4%	56.8%	79.0%	47.5%
Vista proba LiveCD	74.8%	62.2%	91.8%	30.6%
Vista proba 60G	85.6%	79.2%	96.8%	50.6%

Ces résultats montrent que les tables probabilistes sont particulièrement bien adaptées pour casser des ensembles de mots de passe de com-

plexité relativement homogène et proches de ceux de l'ensemble d'entraînement, comme les ensembles RockYou, Yahoo et MySpace. Même sur l'ensemble de test Online, très hétérogène et contenant des mots de passe plus complexes, les tables probabilistes tiennent facilement la comparaison avec des tables classiques, le tout avec une taille très inférieure.

Cela se confirme également en constatant que seuls environ 1% des mots de passe de l'ensemble de test RockYou qui sont cassés par la combinaison Vista num et Vista special XL ne sont pas cassés par les tables Vista proba LiveCD. De manière analogue, seuls environ 1% des mots de passe cassés par la combinaison Vista num, Vista special XL et Vista eight XL ne sont pas cassés par les tables Vista proba 60G.

Ainsi ces tables sont très intéressantes lorsque l'on désire casser un maximum de mots de passe d'un ensemble d'une certaine taille, alors que les tables classiques conviennent mieux au cassage de mots de passe isolés avec un taux de succès de 99% si celui-ci appartient à l'ensemble cible.

Conclusions

Ce travail a montré l'impasse dans laquelle les tables Rainbow "classiques" se trouvaient malgré les avancées en terme de technologie, en particulier les GPU et SSD. Lorsque l'on désire couvrir des mots de passe de longueur plus grande que huit caractères avec un alphabet complet, ces tables sont limitées par leur taille et donc par les temps d'accès au stockage. Ainsi, elles ne sont plus en mesure de suivre la courbe théorique de performance, et les attaques par brute force qui peuvent utiliser pleinement la puissance offerte par les GPU ont repris l'ascendant.

L'approche probabiliste décrite, basée sur une combinaison de patterns et de chaînes de Markov, engendre en particulier deux conséquences importantes. La première est la réduction de la taille de tables tout en conservant un taux de succès très intéressant pour des mots de passe plus complexes. La deuxième réside dans la complexification de la fonction de réduction et donc dans l'augmentation du temps de calcul des hash-réductions. Ces deux conséquences tendent à donner un nouvel équilibre aux tables Rainbow car elles peuvent à nouveau bénéficier pleinement des avancées technologiques récentes, au prix d'une complexité plus importante dans l'implémentation.

Finalement nous avons aussi montré qu'une implémentation efficace en GPU est possible même pour une combinaison de patterns et de chaînes de Markov d'ordre deux si on utilise l'algorithme Alias pour

simuler un dé biaisé et une structure de données compacte pour stocker les probabilités.

Références

1. Mask attack in hashcat. `http://hashcat.net/wiki/doku.php?id=mask_attack_in_hashcat`.
2. Schneider Alain. Rainbow tables probabilistes. *SSTIC*, 2011.
3. Gildas Avoine, Pascal Junod, and Philippe Oechslin. Characterization and improvement of time-memory trade-off based on perfect tables. *ACM Trans. Inf. Syst. Secur.*, 11(4) :17 :1–17 :22, July 2008.
4. Gosney Jeremi. Password cracking hpc. *Password12*, December 2012.
5. Schwarz Keith. Darts, dice, and coins : Sampling from a discrete distribution, December 2011. `http://keithschwarz.com/darts-dice-coins/`.
6. Weir Matthew. *Using Probabilistic Techniques to Aid in Password Cracking Attacks*. PhD thesis, The Florida State University, May 2010.
7. Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 364–372, New York, NY, USA, 2005. ACM.
8. Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Lecture Notes in Computer Science*, volume 2729, pages 617–630. Springer, 2003.
9. Sophos. Security at risk as one third of surfers admit they use the same password for all websites, March 2009. `http://www.sophos.com/en-us/press-office/press-releases/2009/03/password-security.aspx`.
10. Michael D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Softw. Eng.*, 17(9) :972–975, September 1991.