Rainbow Tables probabilistes

Alain Schneider

Groupe LEXSI
Tours Mercuriales Ponant - 40 rue Jean Jaurès
93170 Bagnolet, France
aschneider(@)lexsi.com
alain.schneider(@)ozwald.fr

Résumé De nos jours il est possible de trouver sur le net des rainbow tables, fruits de longs travaux collaboratifs, qui pèsent plusieurs centaines de Go et décrivent des espaces pouvant aller jusqu'à toutes les combinaisons possibles de caractères ascii de longueur 1 à 8. Ces tables ≪ classiques ≫ ont pour principal inconvénient que les espaces de mots de passe parcourus grandissent de façon exponentielle avec la longueur des mots de passe recherchés et qu'atteindre les 9 caractères est, à l'heure d'aujourd'hui, hautement improbable. Ainsi les tables basées sur des dictionnaires et celles dites ≪ hybrides ≫ sont apparues dès 2007 et 2008. Ces deux approches visent à réduire la taille de l'espaces de mots de passe candidats tout en conservant un taux de succès élevé. Ces variations sont actuellement, à notre connaissance, les seules implémentations publiques qui tentent d'améliorer la pertinence des espaces couverts par des rainbow tables. La littérature propose pourtant d'autres axes d'améliorations, notamment statistiques, qui sont très prometteurs. Nous présentons l'une de ces techniques dans cet article et nous en évaluons l'intérêt à travers une preuve de concept. De plus, un outil original implémentant ce mécanisme de rainbow tables statistique est décrit. Les résultats expérimentaux obtenus avec cet outil sont exposés et montrent qu'avec une table de 8Go le taux de succès est équivalent à une table classique de 15To. Des résultats supérieurs, obtenus avec une table statistique de 2To, sont également exposés...

Mots-clés: cassage de mot de passe, GPU, CUDA, Markov, Rainbow Tables

1 Introduction

1.1 Stockage des informations d'authentification

Les problématiques d'identification et d'authentification en informatique se règlent, traditionnellement, à l'aide d'une mécanique basé sur ≪ login ≫ et ≪ mot de passe ≫. Pour que cette mécanique puisse être utilisée il faut que l'utilisateur et le système informatique partagent deux informations :

- Le login : c'est l'élément d'identification. Il est considéré public.
- Le mot de passe : c'est l'élément d'authentification, il doit rester secret, connu seulement de l'utilisateur et du système d'authentification.

La sécurité informatique étant ce qu'elle est, il est souhaitable que le secret partagé (i.e. le mot de passe) reste secret même si le système informatique authentifiant venait

à être compromis. Afin de garantir la confidentialité du mot de passe il est d'usage que le système authentifiant stocke un condensat cryptographique [1] du mot de passe, plutôt que le mot de passe lui-même.

Un condensat cryptographique est le résultat de l'application, à un message clair, d'une fonction dite « de hachage ». Une bonne fonction de hachage doit, entre autre, présenter les propriétés suivantes :

- Il est très difficile de trouver deux messages aléatoires qui donnent le même condensat cryptographique ;
- Il est très difficile, à partir du résultat d'un hachage, de retrouver le message initial auquel a été appliqué la fonction de hachage.

La première propriété assure que la qualité de l'authentification n'est pas dégradée lorsqu'on la base sur une comparaison de hash plutôt que sur une comparaison des mots de passe en clair.

La seconde propriété implique que même si la confidentialité des hashs stockés par le système authentifiant venait à être compromise les mots de passe des utilisateurs resteraient secrets, or c'est cette propriété qui était recherchée.

Bien que l'on trouve encore certains systèmes qui stockent les mots de passe en clair[2] et d'autres où la connaissance du hash peut se substituer à la connaissance du mot de passe dans le protocole d'authentification[3], l'état de l'art stipule de ne stocker que des condensats cryptographiques de mots de passe, et pas les mots de passe en clair.

1.2 Méthodes d'attaque

Stocker uniquement des hashs, et pas les mots de passe en clair, ne garanti pas pour autant la confidentialité de ces derniers. En effet plusieurs types d'attaques peuvent être menés sur les hashs dans le but de retrouver les mots de passe.

Le premier type d'attaque est intuitif : trouver une méthode d'inversion de la fonction de hachage. Plusieurs travaux ont été menés sur le sujet et nous pouvons considérer aujourd'hui qu'il n'existe pas de méthodes d'inversion pour les fonctions de hachage modernes (telles que celles de la famille SHA-2 par exemple).

Le second type d'attaque est relatif aux attaques dites par force brute. On sélectionne des mots de passe candidats, on leur applique la fonction de hachage, et on compare les résultats avec le hash que l'on cherche à inverser. La collision entre un hash généré et le hash recherché nous indique probablement que le mot de passe initial a été retrouvé [2].

1. également appelé « hash »
2. Ou, pour le moins, que l'on a trouvé un mot de passe valide.

2 Bref état de l'art du cassage de mots de passe

Casser des mots de passe à partir de leur hash revient donc, approximativement, à sélectionner un grand nombre de mots de passe candidats, à en calculer le condensat cryptographique, puis à comparer ce condensat avec le condensat que nous cherchons à casser. Cette méthode comporte deux axes d'optimisation immédiats : d'une part le choix des mots de passe à tester, et d'autre part la méthode de calcul des condensats.

2.1 Choix des mots de passe candidats

On peut distinguer deux grandes méthodes classiques pour définir les mots de passe candidats :

- La sélection dans une liste de mot prédéfinis (dictionnaire) ;
- L'énumération exhaustive des combinaisons d'un alphabet, jusqu'à une longueur donnée.

Les attaques par dictionnaire sont très rapides à dérouler, le nombre de mots de passe candidats dépassant rarement le million, et donnent généralement de bon résultats. Il est, de plus, possible d'augmenter le nombre de mots de passe candidats découlant de l'utilisation d'un dictionnaire en appliquant aux mots présents une série de modifications prédéfinies (ajout d'un symbole ou de chiffres en fin de mot, modification de la casse, etc.). Les attaques de ce type permettent donc d'obtenir rapidement des résultats, mais restent limitées en évolution à cause de la contrainte de devoir stocker le dictionnaire qui ne peut donc pas grossir infiniment.

Les attaques par énumération exhaustive consomment souvent plus de temps avant de donner des résultats, mais elles ont l'avantage de pouvoir faire tomber n'importe quel mot de passe « pourvu qu'on laisse tourner l'attaque assez longtemps ».

2.2 Stratégies de calcul

Historiquement, optimiser la stratégie de calcul des hashs se résumait à trouver un algorithme pour CPU qui tournerait plus vite de quelques cycles que l'algorithme précédent. Puis, avec la montée en puissance d'internet en particulier, la philosophie du calcul distribué s'est répandue et on a vu apparaitre de plus en plus de projets similaires à BOINC qui permettent de partager un calcul complexe entre plusieurs machines reliées par un réseau informatique. Mais l'évolution la plus marquante de ces dernières années est le déport de calculs sur d'autres matériels que des CPU.

Les premiers pionniers du calcul sur matériel exotique ont exploré les capacités offertes par des FPGA. Les performances étaient au rendez-vous, mais la programmation de ces équipements peut s'avérer complexe, et leur nature même en faisait un

équipement à réserver aux professionnels. Ce n'est donc qu'avec l'arrivée du calcul sur matériel à vocation graphique grand public que le calcul hors CPU s'est démocratisé. Le premier matériel graphique à avoir beaucoup fait parler de lui en la matière est une console de jeu : la PS3. A titre d'exemple le cluster de quelques 1700 PS3 mis en place par le gouvernement américain dans le projet CONDOR figure toujours parmi les 40 supercalculateurs connus les plus puissants du monde.

La démocratisation du calcul hors CPU se poursuit aujourd'hui avec les projets CUDA et OPENCL qui permettent d'utiliser des cartes graphiques grand public pour du calcul indifférencié (en particulier pour du cassage de mot de passe). C'est actuellement cette voie qui affiche les performances les plus impressionnantes, les gains étant de plusieurs ordres de grandeurs par rapport à du calcul sur CPU.

2.3 Compromis temps/mémoire par Rainbow Tables

Principe général Une table arc-en-ciel (également appelée Rainbow Table) est une structure de données qui permet d'améliorer le compromis temps-mémoire proposés par Martin Hellman dans les années 1980. Ces structures de données ont été inventées en 2003 par Philippe Oechslin[4] et ouvrent des horizons intéressants.

Le principe des Rainbow Tables repose sur la définition d'une fonction dite « de réduction » qui, alliée à une fonction de hachage, permet de définir des chaines composées de plusieurs milliers de maillons constituées comme suit :

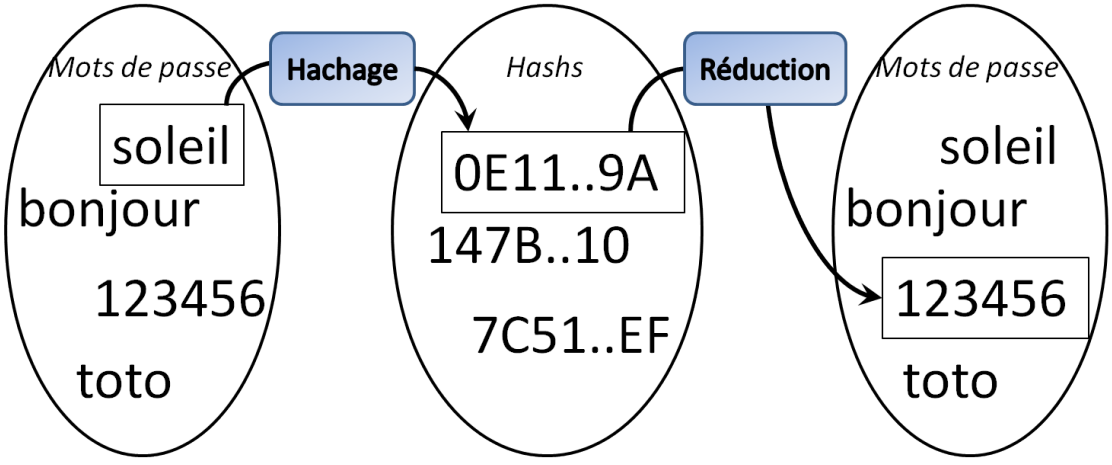


Figure 1. Structure d'un maillon de chaine

Une chaine peut ainsi être représentée par son premier et son dernier élément uniquement, les maillons intérieurs pouvant être reconstruits à la demande grâce à l'application successive des fonctions de hachage et de réduction. C'est grâce à cette astuce de représentation que le compromis temps-mémoire s'opère : en stockant uniquement deux éléments sur disque (le premier et le dernier d'une chaine), on mémorise en fait l'association de plusieurs milliers de mots de passe avec leur hash respectif.

La recherche d'un mot de passe associé à un hash donné s'opère comme suit :

- On vérifie si le hash recherché est l'une des fins de chaines stockées :
  - Si tel est le cas on reconstruit cette chaine à partir de son premier élément à la recherche de la dernière association : mot de passe / hash, ce qui nous donnera probablement un mot de passe qui, une fois haché, correspond bien au hash que nous cherchons à casser.
- Si le hash recherché n'était pas présent dans les fins de chaines que l'on a stockées, on fait alors l'hypothèse qu'il est peut-être en avant dernière position de l'une de nos chaines. On applique donc à ce hash recherché la fonction de réduction puis de hachage, et on cherche le hash ainsi obtenu parmi les fins de chaines stockées :
  - Si le hash obtenu est bien parmi les fins de chaines que l'on a stockées on reconstruit, grâce à son premier élément, la chaine dont il est la fin. L'avant dernière association : mot de passe / hash, nous donnera probablement un mot de passe qui, une fois haché, correspond bien au hash que nous cherchons à casser.
- Si le hash obtenu n'est pas parmi les fins de chaines stockées on fait l'hypothèse que le hash que nous recherchons est peut être dans l'avant-avant-dernier maillons de l'une de nos chaines. On applique alors deux fois le couple « réduction/hachage » au hash que l'on cherche à casser puis on recherche le hash obtenu dans les fins de chaines stockées. On répète ainsi la procédure jusqu'à avoir cassé le hash ou épuisé toutes les positions possibles dans nos chaines.

En utilisant ce principe il est possible de calculer d'avance un grand nombre de hashs pour les stocker dans une Rainbow Table de taille raisonnable et en faire usage rapidement à un moment ultérieur. L'une des conséquences de ce principe est que les Rainbow Tables se prêtent particulièrement bien à un contexte distribué : chaque participant à la génération apporte quelques chaines à l'édifice et pourra profiter du résultat des calculs effectués par tous les autres en quelques secondes une fois la Rainbow Table terminée.

Améliorations publiques de la pertinence Les fonctions de réductions classiques utilisées à ce jour dans les Rainbow Tables opèrent en fait en deux temps. Au lieu de réduire directement un hash en un mot de passe, ces fonctions passent par une étape intermédiaire de réduction en nombre entier. Les opérations de réductions vues précédemment se déroulent donc en réalité de hash vers nombre entier, puis de nombre entier vers mot de passe.

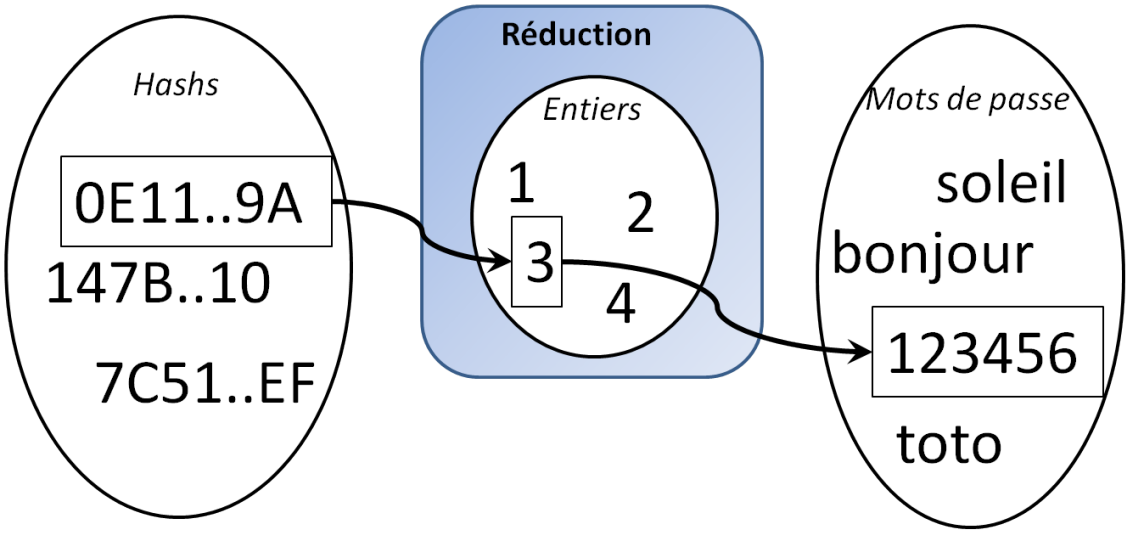


Figure 2. Décomposition d'une fonction de réduction générique

Ce fonctionnement permet d'utiliser comme espace de mot de passe n'importe quel espace énumérable sans avoir à redéfinir des fonctions de réductions. C'est un avantage certain car redéfinir de nouvelles fonctions de réduction n'est pas chose aisée, en effet ces fonctions, en plus de devoir être nombreuses (une fonction différente par maillon), doivent présenter plusieurs caractéristiques statistiques, dont celle d'homogénéité, sous peine d'hypothéquer la qualité des tables générées.

Usuellement les Rainbow Tables décrivent donc des espaces constitués de l'énumération de toutes les combinaisons possible d'un certain alphabet jusqu'à une longueur donnée. Ces espaces permettent en effet d'obtenir très rapidement, à partir d'un entier N donné, le N-ième mot de passe de l'ensemble et ils sont donc parfaitement adaptés à un usage en Rainbow Table. Des variations ont cependant rapidement vu le jour.

En 2007 par exemple une variation qui propose d'utiliser un dictionnaire comme espace de mot de passe voit le jour[5]. Cette variation est possible à moindre frais car les mots de passe contenus dans un dictionnaire sont trivialement énumérables

et on accède toujours rapidement au mot de passe du numéro de son choix. Cette variation sera cependant rapidement oubliée, son intérêt étant limité aux algorithmes excessivement lents à calculer [3].

En 2008 une autre variation voit le jour : l'utilisation d'espaces hybrides[6]. Les espaces parcourus par les Rainbow Tables hybrides sont simplement des produits cartésiens de deux espaces « classiques » constitués de l'énumération de toutes les combinaisons possible d'un alphabet sur certaines longueurs. Grace à de tels espaces on peut décrire des mots de passe longs, constitués d'une variété importante de caractères, tout en conservant un temps de calcul raisonnable. Un exemple de tel espace hybride serait le produit cartésien de l'espace des mots composés d'une unique lettre majuscule avec celui des mots composés de huit lettres minuscules, on obtient alors l'espace résultant des mots de neufs lettres commençant par une majuscule suivi de huit minuscules ; un tel espace est 512 fois plus petit que l'ensemble des mots de neufs lettres majuscules ou minuscules, il est donc beaucoup plus rapide à parcourir tout en étant quasiment aussi pertinent dans un contexte de mots de passe.

Le but de ces deux améliorations est le même : permettre d'utiliser, dans des Rainbows Tables, des espaces de mots de passe plus pertinents qu'une énumération exhaustive des combinaisons d'un alphabet donné sur une longueur donnée. Les travaux dans ce sens se poursuivent et depuis quelques mois le projet freerainbowtables.com propose une nouvelle version des tables hybrides qui permettent de faire le produit cartésien de plus de deux ensembles. Des ensembles du type « majuscule suivi de minuscules suivi de chiffres » sont donc à présent possibles.

Améliorations publiques du compromis temps/mémoire Nous avons vu que la création du maillon d'une chaine se déroule en trois étapes : un hash est réduit en entier, puis cet entier est converti en mot de passe, enfin ce mot de passe est prêt à être haché pour générer le maillon suivant. Un maillon peut donc être représenté soit par son hash, soit par son entier correspondant, soit par le mot de passe qui en est déduit.

De plus nous avons vu que pour stocker une chaine il suffit de stocker son premier et son dernier maillon. Ainsi une chaine peut être représentée de 9 façons différentes. La représentation la plus simple pour une utilisation informatique c'est de stocker l'entier correspondant au premier maillon et l'entier correspondant au dernier, ainsi une chaine est représentée simplement par deux entiers.

Les Rainbow Tables sont donc très souvent échangé sous le format simple d'une suite de couple d'entiers 64 bits. Ce format est aisé à comprendre, et rapide à implémenter, en revanche il est gourmand en espace disque. Pour se rendre compte

3. Peut-être va-t-on revoir ce type de Rainbow Tables avec l'arrivée d'algorithmes tels que MsCache V2.

de l'espace perdu une constatation naïve suffit : compresser en bzip2 des tables sous ce format.

Rainbow Table	Taille originale	Taille bzipée
lm_all-space#17	1,0 Go	616 Mo
md5_alpha-numeric-space#1-8	850 Mo	513 Mo
ntlm_alpha-numeric-space#1-8	1,0 Go	618 Mo
ntml_mkv291#stats.optimize	9,0 Go	3,7Go

Ces gains important de place, trahissant une entropie faible et donc un « gâchis de place », peuvent être partiellement expliqués par les deux considérations suivantes :

- L'espace de toutes les combinaisons possibles de lettres minuscules allant jusqu'à 9 lettres contient un peu plus de 2^{42} mots de passe, c'est-à-dire que l'entier représentant un maillon sera forcément contenu entre 0 et 2^{43}, et donc que 43 bits suffiraient à stocker cet entier. Or dans le format standard des Rainbow Tables ce sont 64 bits qui sont utilisés pour stocker cet entier. Nous avons donc intrinsèquement presque un tiers d'espace utilisé inutilement car il ne contiendra que des bits nuls.
- L'algorithme d'utilisation des Rainbow Table répète de nombreuses fois la recherche d'un entier parmi l'ensemble, très conséquent, des entiers représentant les fins de chaines qui ont été stockées. Il est donc essentiel que ces recherches soient rapides, c'est pourquoi les chaines qui constituent une table sont sauvegardées triées par ordre croissant de l'entier représentant la fin de chaine afin qu'il soit possible d'effectuer des recherches par dichotomie sur cet ensemble. La conséquence immédiate est que deux fins de chaine consécutives sont probablement proches l'une de l'autre et qu'il est donc peu avantageux de stocker ces deux entiers plutôt que de stocker leur écart, ce qui demanderait a priori très peu de bits.

Des améliorations portant sur le format de stockage ont donc vu le jour. On peut par exemple citer le format dit « Compact » [4], qui tire principalement parti du premier point exposé ci-dessus, ou encore les formats « indexés » [5] qui permettent des réductions de taille encore plus impressionnantes en tirant parti des deux points ci-dessus.

En plus des optimisations ne s'intéressant qu'au format de stockage un autre levier de réduction de la taille des fichiers existe et porte, lui, sur le contenu même des tables. Pour bien comprendre cette optimisation rappelons comment sont générées les chaines : d'abord un entier de départ est choisi, généralement aléatoirement, pour

4. Extension .rtc
5. Extension .rti

initialiser le premier maillon, ensuite cet entier est transformé en mot de passe, ce mot de passe est haché, le hash est réduit en entier qui sera le second maillon, et ainsi de suite jusqu'à avoir le nombre de maillon que l'on a défini pour nos chaines. On voit alors que, bien qu'étant liés les uns aux autres de façon déterministe, la suite des maillons peut-être considérée comme pseudo-aléatoire. En effet lors du choix du premier maillon il est impossible de prédire les entiers qui décriront les maillons de toute la chaine, ne serait-ce qu'à cause de l'intervention d'une fonction de hachage dans la détermination des maillons de proche en proche. Lors de la génération d'une table il est donc courant qu'un même mot de passe apparaisse dans plusieurs chaines (puisque chaque création de maillon peut être vue comme un tirage pseudo-aléatoire indépendant dans l'ensemble des mots de passe). Si un même mot de passe apparait dans deux chaines différentes à la même position, de par le déterminisme du mécanisme de création des chaines, les deux chaines se poursuivront forcément de façon identique et aboutiront en particulier au même maillon de fin. Stocker ces deux chaines reviendrait alors à stocker de l'information redondante (toute la fin de chaine depuis le premier mot de passe qu'elles ont en commun). Un axe d'amélioration de la taille des fichiers de Rainbow Tables consiste donc à ne pas conserver les chaines se terminant par un maillon de fin déjà présent parmi les fins de chaines, ainsi on s'assure de ne pas stocker ce type de redondance et la taille de la Rainbow Table s'en trouve réduite. Les tables ainsi optimisées sont appelées « parfaites » et n'ont donc aucune fin de chaine en double. Cet axe d'optimisation a l'avantage de n'impliquer aucune modification des logiciels utilisant les Rainbow Tables, mais il a l'inconvénient majeur de nécessiter beaucoup plus de temps de calcul. En effet beaucoup de chaines seront jetées une fois calculées car elles auront une fin déjà présente, on estime donc que la création d'une table parfaite prend environ 5 fois plus de temps que la création d'une table non-parfaite équivalente. Le gain de taille sur les fichiers, lui, gravite dans les 35%.

Les améliorations sur le format de stockage sont bien entendu cumulables avec la perfection des tables et il est possible de voir des tables équivalentes en couverture mais qui ont des tailles variant d'un facteur 3 ou 4.

3 Filtres Markoviens

3.1 Propriété statistiques des mots de passe

Nous avons vu précédemment que chercher à utiliser des espaces de plus en plus pertinents est un axe important du progrès de l'efficacité des Rainbow Tables. Malheureusement les implémentations publiques sont encore relativement pauvres alors que la littérature a déjà proposé des axes riches qui s'appuient plus finement sur

des considérations statistiques de constitution des mots de passe que ne le permettent les espaces hybrides.

Dès les premières études sur les choix de mots de passe il apparait clair que des séquences purement aléatoires sont difficiles à mémoriser[7] et que les utilisateurs ont donc tendances à choisir des mots de passe dérivés de mots existants ou de moyens mnémotechniques.

Concernant les mots de passe dérivés de mots existants il n'est nul besoin d'être un expert en cassage de code de César par analyse fréquentielle pour se convaincre qu'ils auront des propriétés statistiques communes avec la langue dont ils sont dérivés. En Anglais par exemple la lettre E est environ dix fois plus fréquente que la lettre W, et il est donc clair que cette même lettre E sera environ dix fois plus présente dans les mots de passe dérivés de mots anglais que la lettre W.

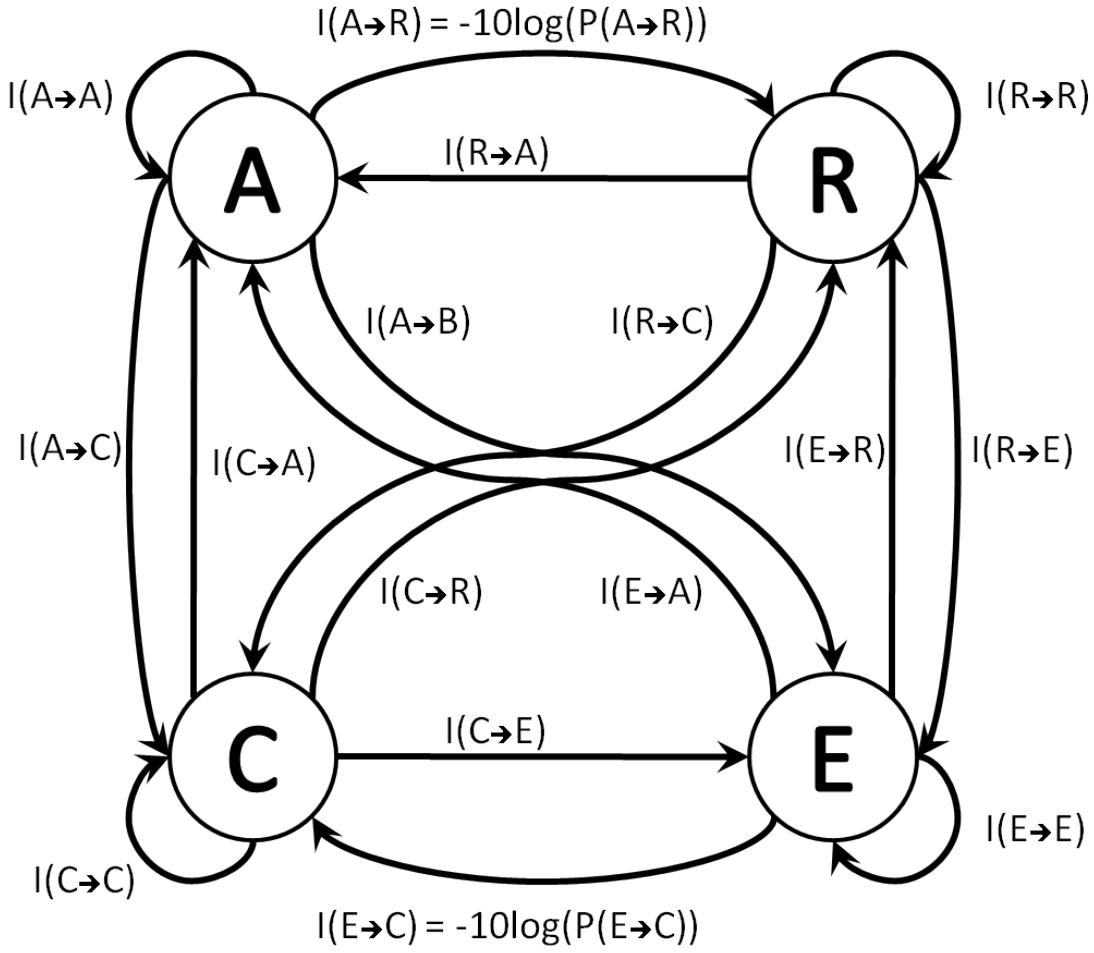
Concernant les mots de passe dérivés de moyens mnémotechniques on retrouvera le même raisonnement. Par exemple dans le cas du moyen mnémotechnique certainement le plus classique et qui consiste à former son mot de passe en utilisant les premières lettres des mots d'une phrase de son choix un biais statistique évident existe. En effet, comme pour les mots de passe dérivés de mots existants, on peut extraire pour chaque lettre, sa fréquence d'apparition en début de mots existants et ainsi déduire des propriétés statistiques des mots de passe générés suivant ce moyen mnémotechnique.

3.2 Automates de Markov

Partant de cette idée que les mots de passe n'étaient pas purement aléatoires mais régis pas des lois statistiques Arvind Narayanan et Vitaly Shmatikov ont formulé, dès 2005, l'hypothèse que l'écriture d'un mot de passe pouvait être modélisée par un automate de Markov à états cachés[8].

Les automates de Markov à états cachés sont des automates à états finis où la probabilité de chaque transition ne dépend que de l'état actuel et des états précédents. Narayanan et Shmatikov ont donc réalisés des automates possédant un état par lettre d'un alphabet de référence, et dont la probabilité des transitions d'un état à un autre (donc d'une lettre à une autre) ne dépendait que de l'état actuel et des états précédents. Ils ont calculé ces probabilités de transition en utilisant une base de mots de passe de référence et ont défini une mesure de l'improbabilité de ces transition valant $I = -10.log(P)$. Fort de ces notations ils ont été en mesure de calculer un score d'improbabilité pour de nouveaux mots de passe, ce score étant la somme des improbabilités des transitions empruntées lors de l'écriture du mot de passe évalué sur leur automate.

Ils arrivent à la conclusion que beaucoup de mots de passe obtiennent un score d'improbabilité très faible lorsqu'ils sont notés par leurs automates alors que des



I(« ARE ») = I(A) + I(A→R) + I(R→E)

I(« AREC ») = I(A) + I(A → R) + I(R → E) + I(E→C)

Figure 3. Automate de Markov

suites de lettres aléatoires ont, elles des scores élevés. De plus ils identifient l'automate qui présente le plus d'intérêt comme étant l'automate dit « d'ordre 1 », c'est-à-dire celui pour lequel la probabilité de transition d'une lettre à une autre ne dépend que de la lettre immédiatement précédente.

Ils établissent également un algorithme, dérivé de l'algorithme de Viterbi, qui leur permet d'énumérer l'ensemble des combinaisons de lettres présentant un score d'improbabilité inférieur à une valeur arbitraire et ils mesurent grâce à cela la taille des espaces que leur automate décrit en fonction de la valeur d'improbabilité maximale fixée. La conclusion en est que, pour un taux de mots de passe cassé avec succès donné, la taille de ces espaces « de Markov » nécessaire est plus petit, de plusieurs ordre de magnitude, que les espaces donnant des taux de succès équivalent décrits par l'énumération exhaustive des combinaisons d'un alphabet sur une certaine longueur.

La pertinence de la méthode est donc démontrée mais il faut attendre 2007 pour voir la première implémentation publique de leur idée. Cette première implémentation est proposée en tant que patch pour le célèbre John The Ripper par Simon Marechal[9]. Le patch rencontre le succès et est actuellement l'un des incontournables de JtR [6].

Aujourd'hui le mode de cassage « incremental » de John The Ripper utilise lui aussi des propriétés statistiques pour générer les mots de passe candidats mais, entre autres différences avec le mode –markov, il remplace l'automate d'ordre 1 par un ordre 2, ce qui rend la probabilité de chaque lettre dépendante des deux précédentes et non plus uniquement de la précédente.

3.3 Algorithme de réduction markovien implémenté dans JTR

L'algorithme implémenté dans John The Ripper colle d'assez près à l'algorithme proposé dans le papier de Narayanan et Shmatikov. L'espace des mots de passe décrit est conceptualisé comme un arbre dont chaque nœud serait caractérisé par : son père, une lettre de l'alphabet, et des transitions vers ses fils. Les transitions vers les fils de chaque nœud est étiquetées avec les probabilités de l'automate de Markov à états cachés. Un mot de passe correspond donc à un cheminement unique dans l'arbre partant de la racine et passant, à chaque étage N, par le nœud étiqueté par la lettre de l'alphabet qui est sa N-ième lettre.

Armés uniquement des probabilités de transitions d'une lettre à une autre servant à définir l'automate de Markov à état cachés, nous sommes en mesure de construire cet arbre de proche en proche. Il nous suffit en effet, pour chaque nœud, de définir quels sont ses nœuds fils, et quel poids attribuer aux branches menant à ces fils. Trouver les nœuds fils revient à sélectionner tous les éléments de notre alphabet qui

6. il est d'ailleurs dans le Jumbo patch

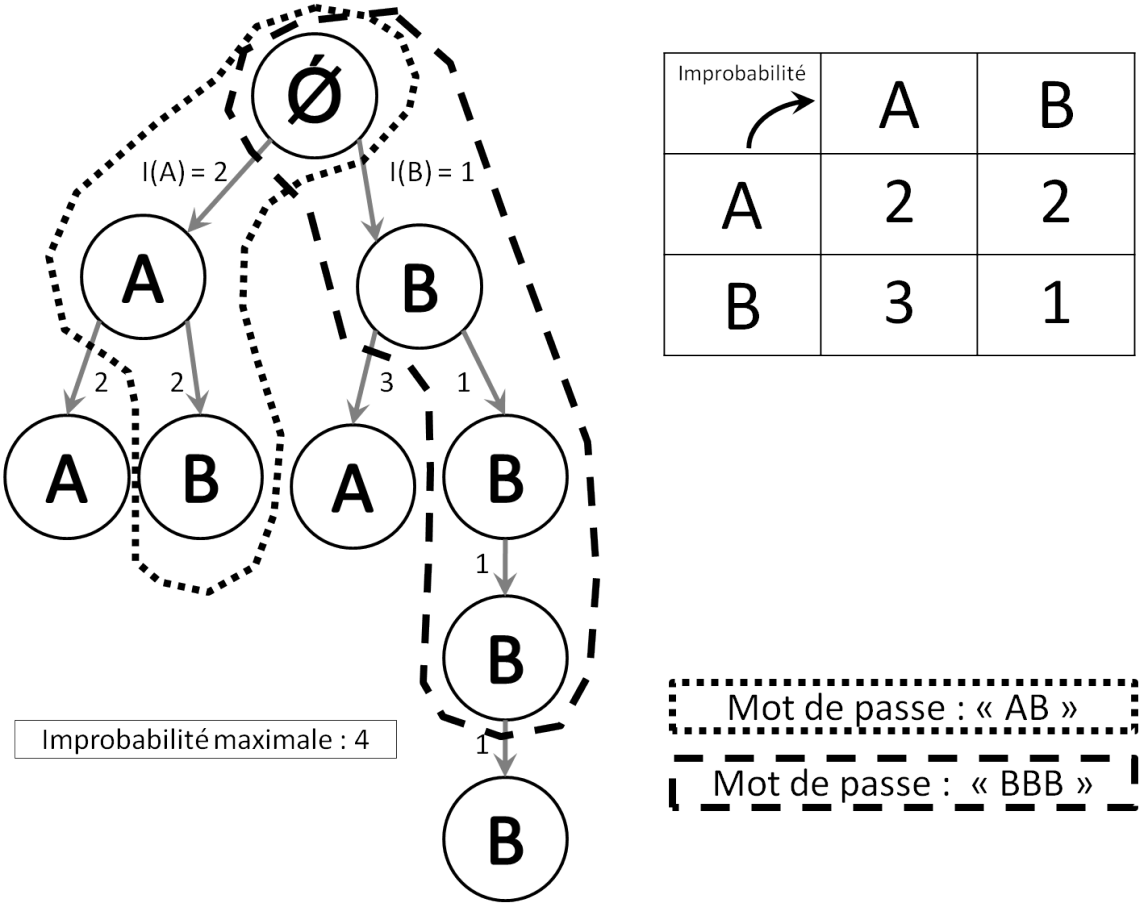


Figure 4. Représentation des mots de passe dans un arbre de Markov

ont une improbabilité de transition depuis la lettre du nœud courant qui soit inférieure au seuil arbitraire d'improbabilité que nous nous sommes fixés. Une fois ces nœuds fils identifiés, attribuer des poids aux branches menant à eux consiste simplement à affecter à chaque branche l'improbabilité de transition entre la lettre portée par le nœud actuel et la lettre portée par son nœud fils. Lorsque nous construirons la suite de l'arbre à partir de l'un de ces nœuds fils il suffira alors de soustraire le poids de cette branche à l'improbabilité arbitraire que nous nous sommes fixée. En construisant ainsi l'arbre lettre par lettre, et en soustrayant à chaque transition le poids de cette transition au capital d'improbabilité que nous nous étions fixé pour construire l'arbre nous arriverons à des nœuds qui ne peuvent avoir de fils, faute d'improbabilité restante suffisante, l'arbre est donc bien fini.

Nous savons à présent construire l'arbre de l'espace des mots de passe de Markov possible, mais les problèmes que nous cherchions à résoudre sont : d'une part de pouvoir dénombrer le nombre de mots de passe que contient cet espace en un temps raisonnable, et d'autre part de pouvoir accéder au N-ième élément de cet ensemble en un temps encore plus raisonnable. C'est en utilisant cet arbre que l'on va pouvoir répondre aux deux problématiques.

Comme l'algorithme de Viterbi, l'algorithme que nous allons utiliser se base sur une constatation d'invariance en fonction des parents. Dans notre cas, si on défini un nœud par son étiquette et un capital d'improbabilité restant, nous pouvons constater que le sous-arbre engendré par ce nœud ne dépend en rien de ses ancêtres. Ainsi deux nœuds distincts dans l'arbre, mais qui portent la même étiquette et le même capital d'improbabilité restant, engendrent exactement le même sous-arbre. Comme dans l'algorithme de Viterbi donc nous pouvons nous servir de cette constatation pour stocker toute l'information de l'arbre dans une structure de données plus compacte car faisant disparaitre cette redondance d'information, et nous pouvons alors nous servir de cette représentation compacte pour résoudre nos deux problèmes.

La structure compacte que nous allons utiliser pour représenter notre arbre, potentiellement énorme, est un simple tableau à deux dimensions. La première dimension de notre tableau défini les étiquettes possibles sur nos nœuds, la seconde dimension défini toutes les valeurs possibles de capitaux d'improbabilité restants.

Chaque cellule de notre tableau représente donc de façon unique un couple « étiquette dans l'alphabet / capital d'improbabilité restant » et nous allons stocker dans cette cellule le nombre de nœuds que compterait un arbre ayant pour racine un nœud portant cette étiquette et ce capital d'improbabilité restant. Nous avons en effet vu, par la propriété d'invariance similaire à Viterbi, que ces deux critères (étiquette et capital d'improbabilité restant), suffisaient à définir n'importe quel sous arbre de notre arbre de Markov, notre arbre de Markov sera donc à coup sûr intégralement représenté

dans notre petit tableau (ainsi que d'autres configurations qui n'apparaissent pourtant pas dans l'arbre mais qui, intrinsèquement, seraient possible et apparaissent grisées dans la figure 5).

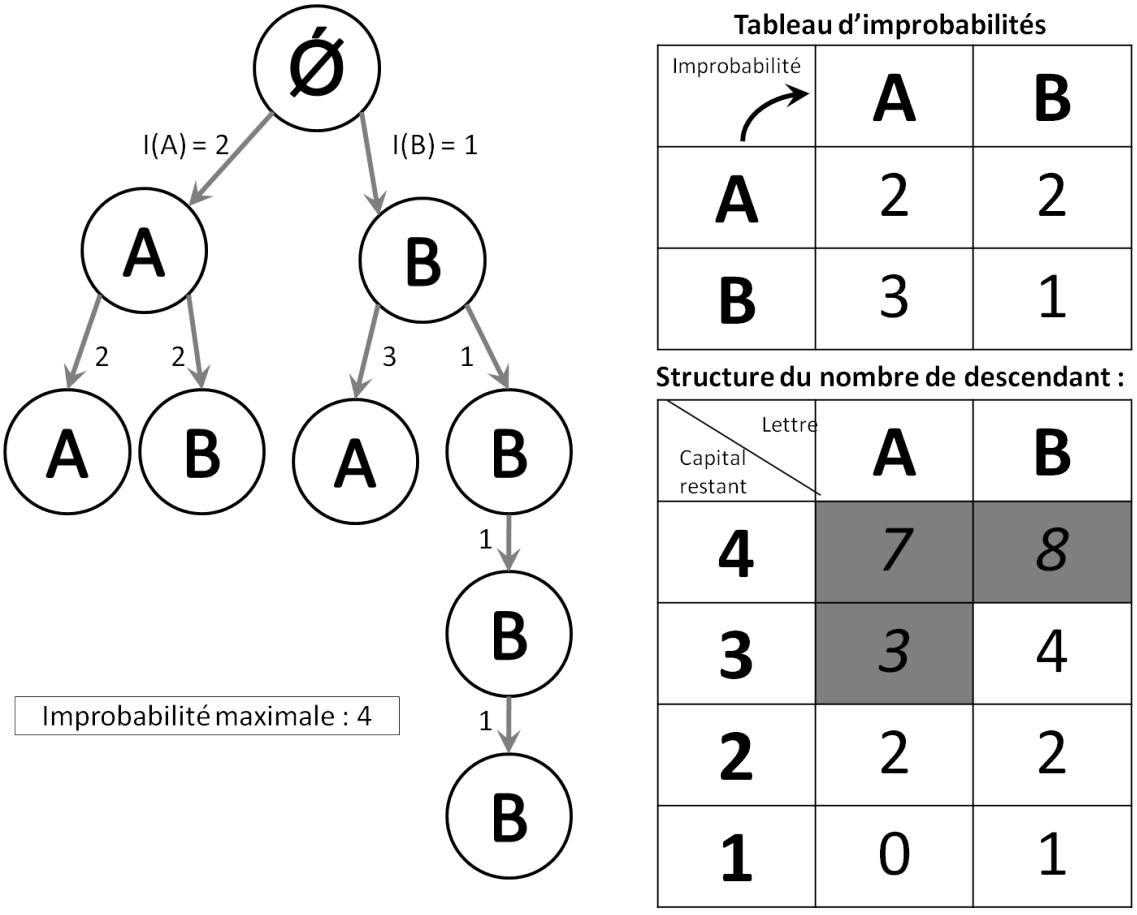


Figure 5. Représentation des mots de passe dans un arbre de Markov

Il nous reste cependant à trouver une façon efficace de remplir ce tableau. Pour ce faire nous procédons de manière récursive simple. Tout d'abord la ligne correspondant au capital d'improbabilité restant nul, par exemple, ne contiendra que des zéros puisqu'avec un capital d'improbabilité nul il est impossible d'emprunter la moindre transition (leur coût étant strictement positif) et donc il est impossible d'avoir le moindre nœud enfant. En partant de ce cas d'initialisation nous remplissons ligne

par ligne notre tableau, par capital d'improbabilité restant croissant. Pour une case quelconque définie par une étiquette E et un capital d'improbabilité restant C, nous savons que le nombre de nœuds descendants compris dans l'arbre ayant pour racine (E,C) est la somme du nombre de ces fils, plus le nombre de nœuds des sous-arbres de chacun de ces même fils :

- Le nombre de fils est aisé à calculer puisque nous savons les identifier en sélectionnant toutes les lettres dont la transition depuis E a un coût d'improbabilité inférieur à notre capital restant C.
- Le nombre de nœuds compris dans l'arbre dont chaque fils est la racine est, lui, déjà connu puisque nous remplissons notre tableaux dans le sens des capitaux d'improbabilité croissant et que ces nœuds fils sont à la racine de sous arbres possédant un capital d'improbabilité restant valant C moins le cout de la transition de (E,C) à eux (coût qui est strictement positif).

Nous sommes donc capables de remplir un tableau qui, pour chaque lettre E de notre alphabet A et chaque capital d'improbabilité restant C, nous donne le nombre de nœuds descendants dans l'arbre ayant pour racine un nœud d'étiquette E et disposant d'un capital d'improbabilité restant C. Cette structure répond à notre premier problème puisqu'en calculant pour chaque lettre E un coup d'improbabilité initial $I_1(E)$ qui correspond à l'improbabilité que cette lettre soit la première d'un mot de passe nous pouvons calculer simplement le nombre de mots de passe d'un espace de Markov regroupant tous les mots possible dont l'improbabilité est inférieur à un seuil arbitraire M :

$$Card(Markov(M)) = \sum_{\forall e \in A, I_1(e) < M} Tableau[e, M - I_1(e)]$$

Notre second problème, qui était celui d'obtenir rapidement le N-ième mot de passe de l'espace pour un N quelconque, est lui aussi réglé grâce à cette structure en tableau. En numérotant les mots de passe de notre arbre comme dans la figure 6 nous pourrons en effet appliquer l'algorithme proposé par Narayanan et Shmatikov et qui résout justement ce problème.

Pour obtenir le mot de passe numéro N nous allons le déterminer lettre par lettre, c'est-à-dire en descendant l'arbre étage par étage. Pour mener à bien cette descente de l'arbre nous conserverons en permanence en mémoire :

- Le capital d'improbabilité qu'il nous reste. A chaque passage d'un nœud à l'un de ses enfants nous déduisons le coût d'improbabilité de cette transition à notre capital, ce qui nous donne en permanence le capital courant dont nous disposons.

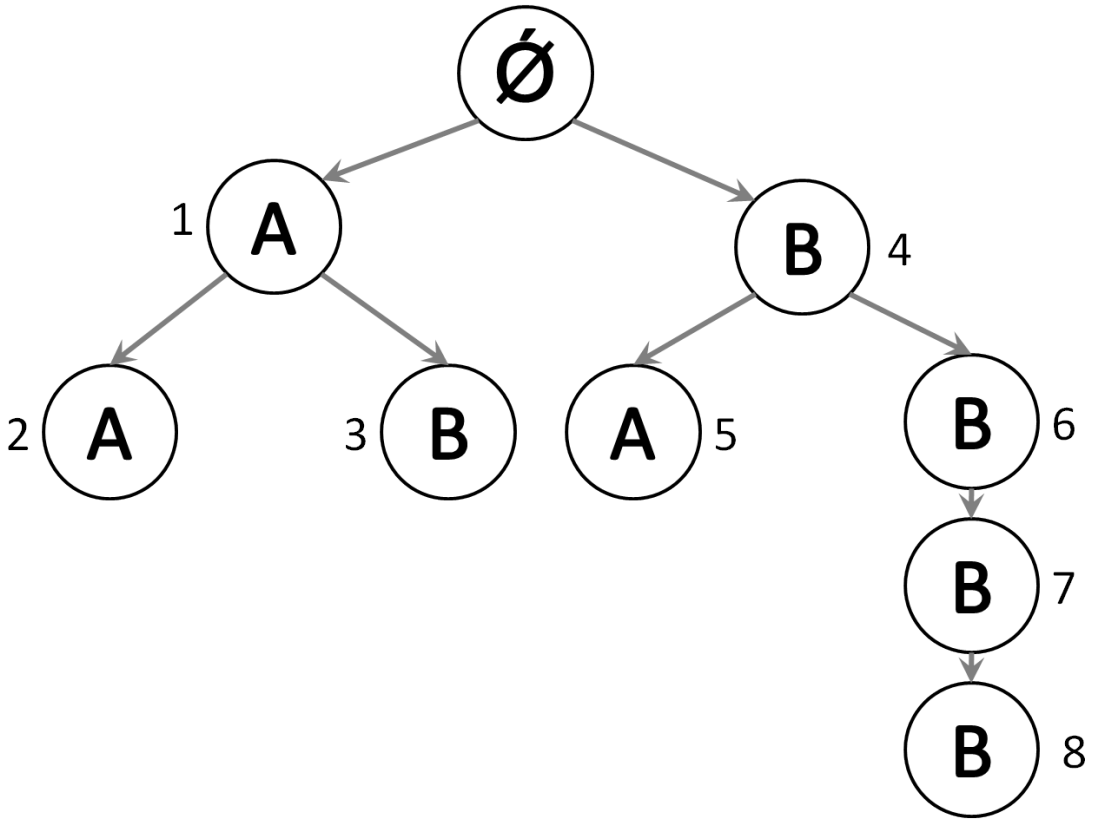


Figure 6. Numérotation des mots de passe dans un arbre de Markov

– Le numéro du mot de passe que nous cherchons, relativement au sous arbre qui aurait notre nœud courant comme racine. Ainsi lorsque l'on descend d'un étage dans notre arbre nous recalculons le numéro relatif de notre mot de passe en lui soustrayant :
  – Le nombre de nœuds que nous avons éliminé à gauche du nœud que nous venons d'élire
  – 1, pour compter le nœud que nous venons d'élire comme porteur de la nouvelle lettre du mot de passe.

La descente se poursuit ainsi jusqu'à arriver à un numéro nul de mot de passe relatif, ce qui signifie que nous avons terminé.

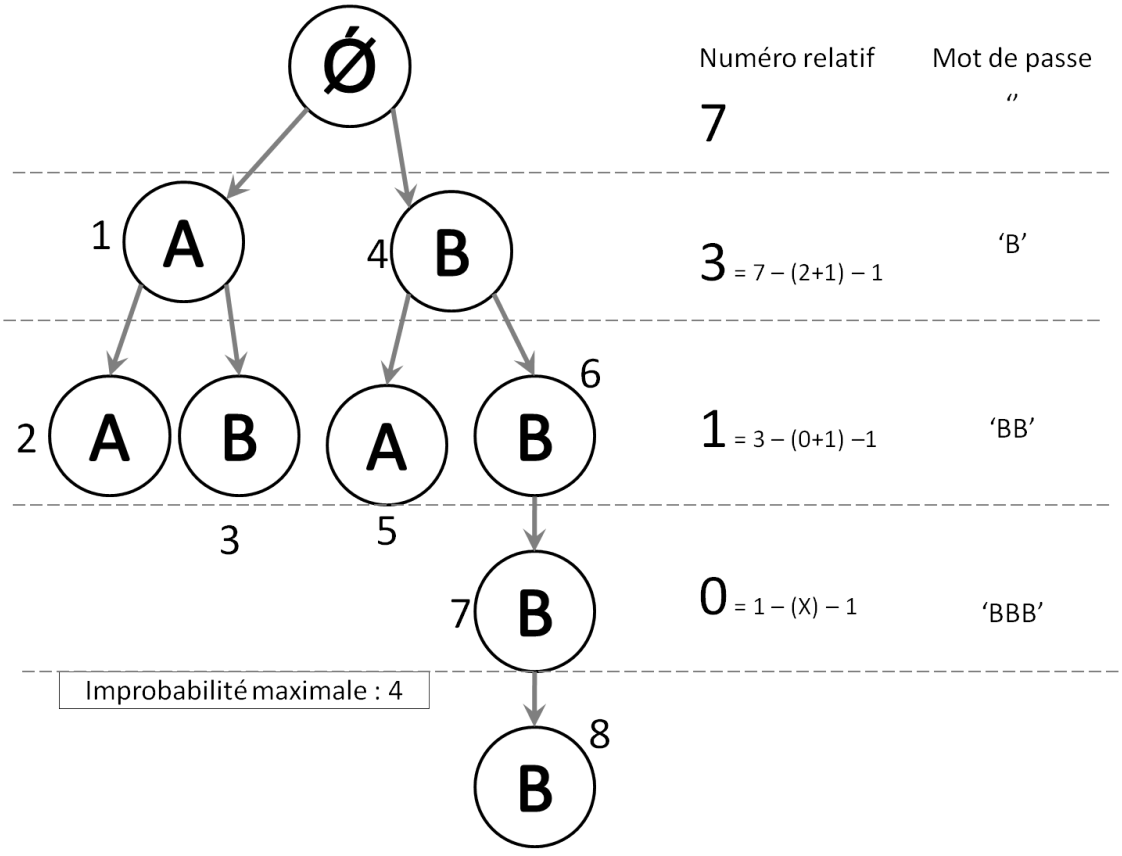


Figure 7. Obtention d'un mot de passe dans un arbre de Markov

En descendant ainsi l'arbre étage par étage on atteint bien le mot de passe numéroté N.

Une amélioration de cet algorithme, implémentée dans le patch de John The Ripper, consiste à parcourir les fils de la dernière lettre connue dans un ordre d'improbabilité croissant plutôt que dans tout autre ordre, ainsi on teste en premier les fils ayant le plus grand nombre de descendants, c'est-à-dire les fils que nous emprunterons le plus probablement. Cette simple amélioration permet de réduire drastiquement les temps de recherche du bon fils, et donc de génération d'un mot de passe d'indice donné.

Pour résumer, l'algorithme de génération du N-ième mot de passe d'un ensemble de Markov se base principalement sur des recherches dans deux structures de données :

- Un tableau à deux dimensions qui, pour chaque couple (A,B) de lettres de notre alphabet, stocke l'improbabilité de la transition de A vers B.
- Un tableau à deux dimensions qui, pour chaque couple (lettre E, capital d'improbabilité restant C), stocke le nombre de descendant d'un arbre de Markov ayant pour racine un nœud étiqueté E avec un capital d'improbabilité restant C.

Nous avons mesuré empiriquement qu'il faut en moyenne tester entre 5 et 6 fils avant de trouver le bon, on peut donc estimer le nombre d'essais nécessaire à la définition d'un mot de passe donné dans un espace de Markov à environ 5,5 fois le nombre de lettre de ce mot de passe.

4 Portage de l'algorithme sur GPU

L'algorithme précédent est applicable sans modification dans un contexte de Rainbow Table, malheureusement les vitesses qu'il permet d'atteindre sont « faibles ». D'après nos tests nous pouvons générer seulement quelques millions de mots de passe par seconde sur un CPU standard, ce même CPU pouvant calculer quelques dizaines de millions de hashs NT dans le même laps de temps. La mode étant actuellement au portage des calculs de hashs sur GPU pour faire des bonds de puissances tout à fait remarquables nous avons alors envisagé de porter l'algorithme de génération de mots de passe de Markov sur GPU dans le but d'obtenir un même bond de performance.

4.1 Présentation de l'environnement CUDA

NVIDIA propose, depuis 2007 pour la toute première version, un kit de développement « CUDA » qui permet d'utiliser leurs GPU pour du calcul dédié. Ce SDK propose une adaptation du langage C qui peut être traité par un compilateur fournit dans le SDK afin de produire des programmes capable de s'exécuter sur GPU.

L'avantage certain du calcul sur GPU est la puissance disponible qui a actuellement largement dépassé ce qui est réalisable sur CPU pour des calculs massivement parallèles. L'inconvénient immédiat du calcul sur GPU, en plus de devoir penser ses algorithmes en parallèle, est le nombre importants de contraintes de développement que doit prendre en compte manuellement le développeur s'il espère obtenir des performances honorables.

4.2 Contraintes mémoire de l'environnement CUDA

Dans cet environnement de calcul massivement parallèle ce sont les accès mémoire qui sont souvent le goulot d'étranglement des performances. En tout cas, ils le sont dans le cas de l'algorithme nous permettant d'obtenir le N-ième mot de passe d'un espace de Markov.

En effet nous avons estimé qu'il nous fallait environ 5,5 étapes de recherches de fils pour chaque lettre d'un mot de passe. Un mot de passe de 8 caractères requiert donc 44 tests de fils. Un test de fils contient lui-même plusieurs accès mémoire :

- Un premier accès mémoire pour savoir quelle lettre est le i-ème fils le plus probable (souvenez vous que nous testons les fils du plus probable au moins probable pour des raisons d'optimisation).
- Un second accès mémoire pour obtenir l'improbabilité de la transition de la lettre courante vers ce fils.
- Un troisième accès mémoire pour obtenir le nombre de descendants qu'aurait ce fils avec le capital d'improbabilité dont il hériterait une fois soustrait le coût de la transition jusqu'à lui.

Définir une seule lettre de notre mot de passe requiert donc en moyenne $5,5 \times 3 = 16,5$ accès mémoire. Ce qui nous amène à la somme impressionnante de 132 accès mémoire en moyenne pour définir un mot de passe de 8 caractères.

Ce ne serait pas un problème dans nos environnements CPU habituels où les accès en mémoires vive ne coûtent pas grand-chose, mais en environnement CUDA c'est un frein majeur. D'une part il faut se rendre compte que ces 132 accès mémoire sont en réalité à multiplier par le nombre de calculs que l'on mène en parallèle (quelques dizaines de milliers dans nos cas pratiques), et d'autre part les accès à la mémoire centrale de la carte graphique d'une carte CUDA souffrent d'une latence que l'on peut qualifier d'astronomique :

« If some input operand resides in off-chip memory, the latency is much higher : 400 to 800 clock cycles. » [1]

Toutes les options possibles pour réduire le nombre d'accès mémoire sont donc bonnes à prendre, nous en verrons une plus loin.

4.3 Contrainte de divergences de threads en environnement CUDA

En environnement CUDA les threads que l'on lance sont, de façon transparente, regroupés par paquets de 32 appelés WARP. Lorsque ces WARPs sont créés les 32 threads le constituant sont irrémédiablement liés les uns aux autres et s'exécuteront en parallèle mais surtout en simultané. En « simultané » signifie qu'aucun thread du WARP ne peut prendre d'avance sur un autre thread du même WARP et qu'ils évalueront donc tous la même instruction au même moment. Que se passe-t-il donc lors d'un embranchement conditionnel ou, par exemple, un seul thread du WARP devrait effectuer une opération de plus que les autres ? Et bien les 31 autres threads se voient attribué un flag « inactif » et les 32 threads du WARP effectuent le calcul supplémentaire en même temps, seul l'unique thread qui n'a pas son flag « inactif » levé prendra alors en compte le résultat de ce calcul, les 31 autres l'ignoreront. On voit bien qu'avec une telle mécanique de WARP il n'est vraiment pas souhaitable d'avoir un thread « à la traine » lors des calculs...

Malheureusement l'algorithme de recherche dans les fils implémentés dans le patch de JtR trouve peut-être le bon fils en 5,5 tentatives en moyenne, mais il présente un fort écart-type. L'écart-type est si fort qu'en moyenne le plus lent des 32 threads d'un WARP a besoin de 32 itérations avant de trouver le bon fils, ralentissant ainsi d'un facteur proche de 6 l'ensemble du WARP par rapport à la vitesse moyenne qu'il aurait eu en environnement CPU.

4.4 Modification de l'algorithme pour environnement CUDA

Une fois les deux contraintes ci-dessus identifiées une modification de l'algorithme s'impose presque d'elle-même. Le cœur de l'algorithme d'écriture du N-ième mot de passe repose sur la recherche du fils chapeautant le sous-arbre qui contient l'indice relatif de notre N-ième mot de passe, cette étape peut donc se ramener à la recherche d'un entier dans une liste d'entiers croissants. En réalisant cette recherche de façon dichotomique plutôt que séquentielle dans l'ordre du plus improbable au moins improbable on ne perd ainsi pas beaucoup en temps moyen, mais on gagne énormément en écart-type (et on verra que l'on gagne également beaucoup en nombre d'accès mémoire).

Cette modification d'algorithme implique par contre une modification inflationniste de notre structure de données principale. En effet nous ne devons plus avoir un tableau qui, à chaque lettre E de notre alphabet et chaque capital d'improbabilité C restant, associe le nombre de fils qu'aurait un arbre de racine (E,C), mais à la place nous devons stocker une liste croissante d'entiers, chacun représentant le nombre de mots de passe compris dans le sous-arbre engendré par le premier fils possible de (E,C),

puis par les deux premiers fils de (E,C), puis par les trois premiers fils de (E,C), et ainsi de suite pour toute la liste. Notre structure de données stocke donc dans chaque « case » une liste aussi longue que notre alphabet, au lieu de stocker un simple nombre. Dans le cas de notre implémentation cela revient à multiplier la taille de cette structure par 128, elle passe donc de valeurs typiques d'environ 10Mo à plus de 1Go.

Grâce à cette modification de méthode de recherche nous avons ainsi borné le pire des threads de nos WARPs à 6 itérations au lieu de 32 en moyenne auparavant, ce qui est un gain important de cycles de calculs. Mais en plus d'avoir gagné en cycle de calcul nous avons également gagné en accès mémoire. En effet pour déterminer le fils adéquat nous n'avons à présent plus qu'à faire les 6 accès mémoire de la recherche dichotomique, puis c'est seulement une fois le bon fils trouvé que l'on regarde dans nos autres structures de données le coût d'improbabilité à soustraire et la lettre de l'alphabet qui correspond à notre choix. Ainsi nous passons de 16,5 accès mémoire en moyenne pour définir une lettre, à seulement $6 + 1 + 1 = 8$. Dans les faits c'est ce phénomène qui s'est avéré prépondérant puisque le passage de la recherche itérative à la recherche dichotomique nous a apporté un gain de vitesse de l'ordre de 100%, ce qui est cohérent avec la division par deux du nombre d'accès mémoire.

4.5 Impact sur les structures de données

Le passage en recherche dichotomique n'a cependant pas que des avantages, en effet nous avons multiplié la taille de notre structure de recherche par 128, ce qui l'amène au dessus du Go. Comme nous avons d'autres structures à faire rentrer dans la mémoire de nos cartes vidéos (comme un buffer où écrire nos mots de passe par exemple...) et que des cartes milieu de gammes n'ont pas plus de 1,5Go de mémoire, il nous a semblé opportun d'investiguer les façons de réduire l'empreinte mémoire de notre programme.

Le premier axe que nous avons adopté est un choix personnel : l'algorithme, tel qu'il est implémenté dans John The Ripper, permet de définir une longueur maximale aux mots de passe que l'on recherche, nous avons décidé de supprimer cette possibilité. La structure principale implémentée dans John n'est en fait pas un tableau à 2 dimensions « lettre de l'alphabet/capital d'improbabilité restant » mais un tableau à 3 dimensions « lettre de l'alphabet/capital d'improbabilité restant/nombre de lettres restante », c'est grâce à cette dimension supplémentaire qu'il est possible de restreindre l'espace de Markov à des mots ne dépassant pas une certaine longueur donnée. Comme nous nous intéressions particulièrement aux mots de passe longs nous avons supprimé cette possibilité, ce qui nous a permis de « perdre » une dimension par rapport aux structures de JTR, et donc de réduire par 19 la taille de notre structure

de données[7]. Grâce à cette astuce nous avons ramené notre structure principale à une taille raisonnable d'environ 40Mo.

Le second axe que nous avons évalué est celui d'une réduction de précision du capital d'improbabilité. En effet notre structure principale a deux dimensions : l'une proportionnelle à la taille de l'alphabet, l'autre proportionnelle aux niveaux de capitaux d'improbabilité. Si notre capital d'improbabilité maximal autorisé est de 20 au lieu de 200, notre structure a donc vu sa taille réduite par 10 car sa seconde dimension va de 0 à 20 au lieu d'aller de 0 à 200. Cet axe fonctionne et nous permet de réduire encore plus la taille de la structure. Si nous étions parvenus à la faire rentrer dans les quelques dizaines de Ko de mémoire rapide que propose NVIDIA sur ses GPU les plus récents nous aurions probablement pu observer un gain de performance important, malheureusement nous ne sommes pas parvenu à réaliser une telle prouesse et notre structure doit donc demeurer en mémoire globale (la zone de mémoire la plus grande) qui souffre d'une terrible latence. Cependant nous gardons cette piste ouverte car en réduisant la taille de cette structure nous observons tout de même de légers gains de performance (certainement dus à une meilleure efficacité du cache de mémoire globale présent en mémoire rapide) et nous n'avons pas observé une baisse de pertinence dans les espaces de Markov à précision réduite par rapport aux espaces de Markov normaux.

4.6 Conclusion partielle

Le portage sur GPU s'est avéré un succès et nous sommes parvenus à obtenir un gain très net de vitesse par rapport à du calcul sur CPU. En effet nous arrivons à calculer une petite dizaine de millions de hashs NT par seconde sur une GTS 450, et environ 40 millions par seconde sur une GTX 580. Ces performances nous ont permis de générer des Rainbows de Markov opérationnelles dont nous exposons les résultats dans la partie suivante.

5 Résultats expérimentaux

LEXSI réalise de nombreux tests d'intrusion chaque année et a donc accès à d'énormes listes de hash de mots de passe réels. Nous avons ainsi disposé d'une liste de plus de 500 000 mots de passe réels et représentatifs. En plus de cette liste de 500k mots de passe réels nous avons obtenu la liste des mots de passe du site web Rockyou qui avait été publiée sur internet en 2009. Pour mémoire le site web stockait les mots

7. Comme nous nous intéressons aux mots de passe long nous mettions systématiquement 19 comme longueur maximale, donc la dernière dimension du tableau allait de 1 à 19.

de passe de ses utilisateurs en clair et, suite à compromission, plus de 32 millions de mots de passe s'étaient ainsi vus publiés sur internet.

Nous utiliserons ces deux listes de mots de passe pour estimer l'efficacité des méthodes de cassages dans cette section en comptant quel pourcentage de chacune de ces listes chaque méthode serait parvenu à casser.

5.1 Dans un contexte de portabilité

Dictionnaires Les attaques par dictionnaires s'adaptent bien à un contexte de mobilité. En effet l'espace requis pour les stocker est en général modeste (quelques Mo, voire quelques dizaines de Mo) et ils permettent quasiment à coup sûr de retrouver des mots de passe, pour peu que l'on ait un nombre conséquent de hash à casser.

Les résultats obtenus par dictionnaire dépendent par contre énormément du contenu des dictionnaires et des règles de mutations appliquées, et nous ne pouvons donc pas présenter de statistiques sur l'efficacité de ces méthodes. Nous vous demanderons donc de nous croire sur parole sur ce point : des dictionnaires classiques ne permettent pas d'espérer plus de 50% de succès sur des échantillons aussi grand que les nôtres.

Force brute L'énumération de toutes les combinaisons possibles d'un certain alphabet s'adapte bien à un contexte de portabilité car il ne requiert quasiment aucun espace disque, mais il présente l'énorme inconvénient de demander un temps non-négligeable avant de donner des résultats. Dans un contexte de mobilité, où il n'est pas rare que l'on ne dispose que de peu de temps, cet inconvénient peut alors être vu comme un défaut majeur. En nous basant sur une hypothèse de 30 millions de hash calculés à la seconde (ce qui est atteignable sur un ordinateur portable standard) nous avons calculé les taux de succès atteignables en environ une journée.

Espace parcouru	Temps requis	Rockyou	500k
LowerAlphaNum 1-7	1h	46%	9%
LowerAlphaNum 1-8	26h	64%	49%
MixedAlphaNum 1-7	32h	48%	13%
LowerAlpha 1-9	50h	35%	6%

Les deux échantillons présentent des différences de taux de succès important et nous ne pouvons que supposer de la cause. Notre hypothèse la plus probable est que la différence importante de résultats tient à la provenance des deux ensembles de mots de passe. En effet les mots de passe de notre ensemble proviennent d'entreprise, ce ne sont donc pas des mots de passe « personnels » comme ceux issus de Rockyou,

leur complexité est donc potentiellement supérieure (on peut en particulier sentir l'influence de la politique de « complexité microsoft » avec le score du charset « LowerAlphaNum 1-8 »).

Petites Rainbow Tables classiques Au vu des résultats assez corrects obtenus par force brute l'utilisation de rainbow table semble tout à fait légitime. En effet le temps peut être une ressource précieuse dans un contexte de mobilité, et les Rainbow Tables permettraient d'obtenir des résultats identiques aux attaques par force brute, mais quasiment instantanément. Nous avons donc sélectionné certaines Rainbow Tables utilisable en contexte de mobilité (i.e. : d'une taille avoisinant la poignée de Go), et relatons leur taux de succès ci-dessous [8] :

Espace parcouru	Taille de la rainbow	Rockyou	500k
LowerAlphaNum 1-7	0,3 Go	46%	9%
LowerAlphaNum 1-8	12 Go	64%	49%
MixedAlphaNum 1-7	15 Go	48%	13%
LowerAlpha 1-9	23 Go	35%	6%

Petites Rainbow Tables de Markov Dans un contexte de mobilité la taille d'une Rainbow Table est donc un élément discriminant, cette taille étant proportionnelle à l'espace de mot de passe décrit on peut espérer obtenir des résultats particulièrement intéressants avec des Rainbow Tables de Markov puisque leur but premier est justement de décrire les espaces les plus pertinents possibles. Nous avons donc sélectionné trois limites de complexité de markov qui décrivent des espaces aboutissant à des rainbows tables de la taille souhaitée (quelques Go) et nous présentons leur taux de succès ci-dessous .

Espace parcouru	Taille de la rainbow	Rockyou	500k
Markov 265 optim	1,04Go (4x267M)	71%	58%
Markov 265	1,11Go(4x283M)	75%	50%
Markov 285 optim	7,99Go (4x2,00G)	79%	69%
Markov 285	8,50Go (4x2,13G)	83%	60%
Markov 300 optim	36,76Go (4x9,19Go)	84%	76%
Markov 300	39,19Go (4x9,80Go)	87%	66%

8. le suffixe "optim" désignant l'utilisation d'un fichier de statistiques que nous avons réalisé sur nos propres échantillons, l'absence du suffixe désignant l'utilisation du fichier "stats" standard fourni avec JTR.

5.2 Dans un contexte de plateforme lourde spécialisée

Dans cette partie nous envisageons le cas d'une plateforme lourde spécialisée dans le cassage de mot de passe (type cluster de calcul). Pour calculer les taux de succès présentés ci-dessous nous nous sommes basés sur une plateforme ayant des capacités proches de la plateforme dont dispose LEXSI, à savoir quelques To d'espace disque disponible, et une capacité CPU d'environ 1 milliards de hash NT calculés à la seconde.

Force brute Bien qu'une attaque par force brute ne soit pas le scénario le plus probable dans le cadre d'une plateforme dédiée (l'investissement dans la génération d'une rainbow table étant largement rentable pour des plateformes amenées à lancer plus de 10 campagnes de cassage au cours de leur vie), nous avons calculé les taux de succès qui seraient obtenus pour diverses campagnes de cassage d'une durée d'environ un mois.

Espace parcouru	Temps de calcul	Rockyou	500k
MixedAlphaNum 1-8	2 jours	68%	63%
LowerAlphaNum 1-10	42 jours	83%	55%
LowerAlpha 1-11	42 jours	39%	7%

Grosses Rainbow classiques L'espace disponible sur une plateforme dédiée peut être compté en To, nous avons donc sélectionné des espaces décrits par des Rainbows Tables de cet ordre de grandeur.

Espace parcouru	Taille de la Rainbow	Rockyou	500k
MixedAlphaNum 1-8	900 Go	68%	63%
LowerAlphaNum 1-10	15 000 Go	83%	55%
LowerAlpha 1-11	15 000 Go	39%	7%
AllChars 1-8 [9]	?	69%	68%

Grosses Rainbows de Markov Concernant le choix des Rainbows de Markov nous ajoutons, en plus de la limite en taille de quelques To, une limite en temps de calcul nécessaire à la génération qui soitde l'ordre de quelques mois tout au plus. En effet le calcul de rainbows de Markov est beaucoup plus lent que le calcul de Rainbow classique, et il n'existe pas non plus de projet de génération distribué participatif. Nous ne considérons donc ci-dessous que des Rainbow Tables que nous savons calculables sur l'infrastructure que nous avons montée (la taille considère des Rainbow Tables imparfaites).

Espace parcouru	Taille de la Rainbow	Rockyou	500k
Markov 315 optim	270Go (4x67Go)	87%	81%
Markov 315	286Go (4x71Go)	89%	71%
Markov 335 optim	2 To (4x512Go)	91%	86%
Markov 335	2,15To (4x550Go)	92%	78%
Markov 350 optim	9,24To (4x2,3To)	93%	88%
Markov 350	9,90To (4x2,5To)	94%	81%
Markov 375 optim [10]	?	95,1%	92%
Markov 375 [11]	?	95,4%	85%

6 Conclusion

Dans un cadre de mobilité où l'espace et le temps disponibles sont faibles l'approche des Rainbow Tables markovienne s'avère particulièrement pertinente. Les résultats obtenus par une Rainbow Markovienne de 1Go dépassent en effet très nettement les taux de succès des Rainbows Tables classiques de taille proche de 10Go.

Dans le cadre d'une plateforme de cassage dédiée les Rainbow Tables de Markov offrent également une alternative très intéressante aux méthodes de cassage classiques et permettent d'atteindre des taux de succès qui ne sont pas atteignables avec les méthodes classiques non combinées.

Références

1. CUDA C Programming Guide
2. Compromission du site web Rockyou et divulgation de 32 millions de mots de passe, stockées en clair... http://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/
3. Toolkit permettant l'utilisation d'un hash NTLM pour s'authentifier dans un domain Windows... http://oss.coresecurity.com/projects/pshtoolkit.htm
4. http://lasecwww.epfl.ch/philippe.shtml
5. Rainbow Tables basées sur dictionnaire http://sites.google.com/site/reusablesec2/drcrack
6. Annonce sur FD du lancement des Rainbow Tables s'appuyant sur des espaces hybrides http://seclists.org/fulldisclosure/2008/Feb/361
7. Jianxin Yan, Alan Blackwell, Ross Anderson,Alasdair Grant
   The Memorability and Security of Passwords –Some Empirical Results
8. Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff
   Arvind Narayanan and Vitaly Shmatikov,
   The University of Texas at Austin
9. http://www.openwall.com/lists/john-users/2007/09/15/2