Mesure de la robustesse des mots de passe

Mathieu Valois

Normandie Université

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité INFORMATIQUE

Préparée au sein de l'Université de Caen Normandie

Mesure de la robustesse des mots de passe

Présentée et soutenue par
Mathieu VALOIS

Thèse soutenue publiquement le 04/12/2019 devant le jury composé de		
Mme CAROLINE FONTAINE	Directeur de recherche au CNRS, École normale supérieure Paris-Saclay	Rapporteur du jury
M. BENJAMIN NGUYEN	Professeur des universités, Institut National Sciences Appliquées	Rapporteur du jury
M. GILDAS AVOINE	Professeur des universités, Université Rennes 1	Membre du jury
M. PATRICK LACHARME	Maître de conférences, ENSICAEN	Membre du jury
Mme MARYLINE LAURENT	Professeur des universités, Inst Nat Telecomm Evry Int Management	Membre du jury
M. JEAN-MARIE LE BARS	Maître de conférences HDR, Université Caen Normandie	Directeur de thèse

Thèse dirigée par JEAN-MARIE LE BARS, Groupe de recherche en informatique, image, automatique et instrumentation

UNIVERSITÉ CAEN NORMANDIE

GREYC

Remerciements

La thèse est une formation à la recherche par la recherche. Le thésard est la personne qui maîtrise le mieux son sujet de thèse, ce qui le force à devoir vulgariser et expliquer auprès de ses collègues, de sa famille et de ses amis ce qu'il fait dans son bureau toute la journée. On entend souvent dire que la thèse est une période difficile et épuisante de la vie d'un chercheur. Je crains le retour de bâton en disant que pour moi, ce fut [1] une expérience épanouissante, d'accumulation de savoirs et de partage.

Même si durant certaines périodes, les songes à l'abandon s'installent, ils ne restent heureusement qu'éphémères. Il est, selon moi, vital de s'évader de son travail de recherche avec de l'enseignement, des loisirs ou des rencontres.

Partager, que ce soit lié ou non au sujet de sa thèse, reste une valeur importante à mes yeux. C'est pour cela que je partage avec vous le fruit de ces trois ans de recherche, compilé dans ce manuscrit.

Je voudrais remercier tout d'abord mes directeurs de thèse, MM. Patrick Lacharme et Jean-Marie Le Bars, pour leur disponibilité, leur patience et surtout leur implication dans cette thèse. J'ai appris grâce à vous que faire de la bonne recherche s'effectue en révisant constamment ses acquis, qu'il faut être patient et persévérant car rien n'est facile du premier coup. Je remercie également les rapporteurs, Caroline Fontaine et Benjamin Nguyen, d'avoir rapporté le manuscrit de manière sérieuse et détaillée, ainsi que les membres du jury, Maryline Laurent, Gildas Avoine, et Iwen Coisel d'avoir accepté notre invitation.

Je remercie la région Normandie pour nous avoir fait confiance, mes directeurs de thèse et moi, sur l'accomplissement des objectifs fixés initialement dans le sujet. Le financement de thèses doit perdurer car il est nécessaire à la formation de chercheurs universitaires, moteurs du progrès scientifique et des avancées technologiques. Il faut que la science française reste de qualité !

Je remercie le laboratoire GREYC de m'avoir accueilli et hébergé, d'avoir fait en sorte que mon travail s'y passe bien et également de tout mettre en œuvre pour que les thèses s'y déroulent dans les meilleures conditions possibles. Une mention spéciale aux porteurs de projets, sans qui maintes publications scientifiques internationales ne pourraient pas être défendues pour faute de financement.

1. ah, ce fût !

Je remercie l'école ENSICAEN de m'avoir hébergé dans ses locaux, de m'avoir accepté au poste d'ATER, et je remercie de plus les responsables de la spécialité informatique de m'avoir confié des enseignements auprès des futurs ingénieurs. Je remercie aussi les responsables des enseignements pour leurs efforts à faire en sorte que les TD/TP se déroulent dans de très bonnes conditions pour les vacataires doctorants et ATER (avoir les corrections de TP permet d'être cohérent et fait gagner un temps fou !).

Je remercie mes collègues de l'équipe Monétique & Biométrie pour m'avoir accueilli dans une bonne ambiance de travail ! Ainsi que mon collègue de bureau, Denis, pour m'avoir accompagné dans les démarches administratives dont nous mériterions d'obtenir un doctorat (voire une HDR), et pour m'avoir accompagné à dénoncer les services qui stockent les mots de passe en clair !

Je remercie ma famille pour avoir été présente dans les moments de fête et de célébration ainsi que dans les moments de partage familial.

Je remercie mes amis, Ghali, Peter, Tanguy, Mark, Anthony, Poulos, pour m'avoir également aidé à profiter des loisirs de la vie, d'avoir poutré du noob, braqué des banques et décimé des dragons, tout ça en ligne. Je remercie les membres du club NTT de m'avoir impliqué dans la pratique sportive et compétitive du tennis de table, et d'avoir fait de moi le meilleur joueur du GREYC (j'attends toujours le tournoi).

Enfin, le meilleur pour la fin, je tenais à dédicacer ce manuscrit aux trois femmes qui m'ont soutenu durant ces trois ans, mes filles Freya et Emma, et ma femme Mélanie. Ton implication dans l'éducation de nos filles ma permis de réaliser ma thèse dans des conditions sereines et je t'en remercie. Je vous remercie toutes les trois de m'avoir forcé à m'évader de cette thèse et de m'avoir apporté autant de joie et de bonne humeur dont j'étais heureux d'être immergé en vous retrouvant le soir.

À vous trois.

Résumé

À l'ère où notre identité numérique se confond toujours davantage avec notre identité personnelle, les besoins en sécurité de nos comptes en ligne sont d'autant plus marqués. Les mots de passe sont à la fois la manière de s'authentifier la plus utilisée et à la fois le maillon le plus faible de la chaîne de sécurité. Malgré l'indéniable fragilité de la plupart des mots de passe utilisés en ligne, le mot de passe reste le meilleur moyen de s'authentifier réunissant sécurité, accessibilité et respect de la vie privée.

L'objectif de cette thèse est de faciliter la conception de mesures de robustesse des mots de passe qui soient pertinentes vis-à-vis des attaques les plus sophistiquées sur les mots de passe. Ces attaques reposent sur des modèles probabilistes de la manière dont les utilisateurs choisissent leur mot de passe. Il s'avèrent que ces attaques sont très efficaces pour trouver des mots de passe plus complexes que ceux habituellement trouvés par les méthodes naïves. Ce travail repose sur trois contributions pour identifier les points clés d'une mesure de robustesse des mots de passe moderne. La première contribution modélise le processus d'attaque sur les mots de passe, formalise et mesure la performance d'un tel processus. La deuxième contribution se charge de montrer que les méthodes actuellement déployées pour mesurer la robustesse des mots de passe ne sont pas suffisantes pour se prémunir des attaques sophistiquées. La troisième contribution analyse algorithmiquement les attaques sophistiquées en observant leur comportement dans le but de concevoir des méthodes efficaces pour augmenter le coût d'exécution de ces attaques. La validation des méthodes s'est effectuée en utilisant des mots de passe issus de fuites de données publiques, totalisant plus de 500 millions de mots de passe.

Table des matières

Introduction

"Sésame, ouvre-toi !" est la phrase permettant aux quarante voleurs d'ouvrir la porte de la caverne dans laquelle ils stockent leur butin, dans le conte "Ali Baba et les Quarante Voleurs" de "Les mille et Une nuits". Cette phrase célèbre sert de mot de passe pour authentifier les personnes ayant le droit d'entrer dans la caverne, elle est une donnée d'authentification, qui peut être partagée (dans le cas des Quarante voleurs) ou bien personnelle. De nos jours, les mots de passe servent à nous authentifier afin d'ouvrir les portes de notre identité numérique, essentiellement en ligne. Le développement d'Internet, de ses applications personnelles et professionnelles ont suscité le besoin que ces mots de passe deviennent des secrets personnels afin de servir à l'authentification d'un unique internaute.

L'utilisation d'un mot de passe est aujourd'hui le moyen d'authentification le plus utilisé sur Internet. D'après Troy Hunt, développeur en sécurité et auteur de "Have I Been Pwned" [22], "*Despite its many flaws, the one thing that the humble password has going for it over technically superior alternatives is that everyone understands how to use it. Everyone.*" [23]("Malgré ses faiblesses, la seule chose qui permette au simple mot de passe de dépasser les alternatives techniques nouvelles est le fait que tout le monde sait comment s'en servir. Absolument tout le monde."). C'est sa simplicité qui fait son succès, mais cela en fait également sa faiblesse. En 2018, Dashlane, une société proposant des services de stockage des mots de passe, estime que l'internaute moyen possède plus de 200 comptes en ligne avec mots de passe [9]. Il est donc évident qu'il ne peut pas retenir ses 200 mots de passe différents, ce qui l'amène bien souvent à les réutiliser. La même année, Chun Wang [50] observe que 52% des utilisateurs sur les 30 millions étudiés réutilisent au moins un mot de passe sur plusieurs services, alors qu'en 2014 des chercheurs estimaient ce nombre entre 43% et 51% [8]. De même, d'autres chercheurs confirmaient que les utilisateurs réutilisaient chaque mot de passe sur entre 1 et 4 sites différents [52].

La réutilisation et la faiblesse des mots de passe ont suscité l'intérêt des pirates informatique. Le site "*haveibeenpwned.com*" [22] recense plus de 340 sites web piratés entre 2007 et 2019, totalisant plus 6 milliards de comptes utilisateurs concernés. Chaque année, toujours plus de services en ligne se trouvent mis à nu et leur données fuitent, contenant bien souvent les mots de passe de leurs utilisateurs. Ce qui inquiète également, c'est la taille de ces bases de données : les plus larges d'entre elles peuvent

contenir plusieurs centaines de millions de comptes, et bien souvent les mots de passe ne sont pas protégés avec les fonctions de hachage adaptées.

Pour palier ce vol massif des mots de passe, il existe une mesure très intéressante : l'authentification forte. Le principe consiste à multiplier les facteurs nécessaires pour authentifier un utilisateur. En utilisant l'authentification forte, le vol du mot de passe ne suffit plus pour se connecter au compte d'un utilisateur, il faut aussi lui voler son téléphone ou son empreinte digitale. Il existe 3 facteurs d'authentification : la connaissance d'un secret, la possession d'un objet et la biométrie. Les mots de passe font partie du facteur connaissance d'un secret, il faut alors prouver la possession d'un objet ou utiliser la biométrie pour parler d'authentification forte avec les mots de passe. Ainsi, il est devenu courant de trouver des services web qui propose l'authentification forte en demandant des modalités de plusieurs facteurs différents, par exemple un mot de passe et la vérification d'un code envoyé par SMS (même si c'est déconseillé à cause des failles dans le protocole SS7 [13]), ou un code généré en fonction du temps (one-time password) via une application sur téléphone. Cependant, la confirmation par email repose sur la connaissance d'un secret et n'offre pas d'authentification forte avec les mots de passe, elle est donc à proscrire dans ce cadre.

Présentation du sujet

La vocation de cette thèse est de proposer, ou préparer des pistes pour concevoir une mesure de robustesse des mots de passe qui soit capable de les protéger des nouvelles attaques basées sur les énumérateurs probabilistes. Pour cela, il est nécessaire de résoudre au préalable deux challenges. Le premier challenge consiste à être capable d'évaluer la performance d'un attaquant qui utilise ces énumérateurs probabilistes pour retrouver les mots de passe d'une base ciblée. Le deuxième challenge consiste à montrer que les méthodes actuellement déployées pour mesurer la robustesse des mots de passe ne sont pas suffisantes pour se protéger en pratique des attaques utilisant ces énumérateurs probabilistes. Ce deuxième challenge justifie le fait qu'il faille concevoir de nouvelles mesures de robustesse qui soient capables de protéger efficacement les mots de passe des énumérateurs probabilistes.

Cette thèse s'intitule **mesure de la robustesse des mots de passe**. Ce titre comporte deux notions qui nécessitent d'être définies : robustesse et mot de passe.

La communauté scientifique s'accorde à dire qu'un mot de passe est une chaîne de caractères secrète qui permet de déverrouiller l'accès à des données. Un mot de passe peut prendre plusieurs formes et être utilisé dans différents contextes : compte local à un ordinateur, compte sur un service en ligne, code PIN d'un téléphone, ...

La robustesse d'un mot de passe est définie comme étant sa résistance à un attaquant qui tente de le deviner. Elle s'exprime en une quantité d'effort qu'un attaquant va devoir rassembler pour le trouver. Selon les usages, cette résistance n'est pas considérée de la même manière : un mot de passe local à un ordinateur fait plutôt l'objet d'attaques ciblées contrairement aux mots de passe des comptes en ligne qui font plutôt l'objet d'attaques massives. La robustesse peut être définie dans trois scénarios différents : la robustesse d'un mot, la robustesse d'un ensemble

de mots, et la robustesse d'un mot dans un ensemble de mots. Il apparaît donc que même si la notion de robustesse est simple à définir, elle est bien plus compliquée à mesurer.

Il existe dans la littérature plusieurs familles de contributions qui proposent une manière de mesurer la robustesse d'un mot de passe en utilisant différents modèles : les entropies, les mesures heuristiques, les chaînes de Markov, les grammaires probabilistes... Weir montre [53] que l'entropie telle que proposée dans les consignes du *NIST Special Publication 800-63-1* [6] n'est pas une bonne mesure de la robustesse. En effet, un mot de forte entropie peut être très populaire et donc être trouvé dans les premiers. C'est pourquoi des modèles probabilistes ont été proposés pour la mesure de la robustesse. Ces modèles nécessitent d'être entraînés sur une base d'apprentissage afin d'obtenir une approximation de la façon dont sont choisis les mots de passe. Il est ensuite possible de calculer, pour un mot de passe donné, la probabilité qu'il soit choisi par un nouvel utilisateur dans ce modèle. Ce qui permet de définir la robustesse du mot en fonction de cette probabilité : moins le mot est probable, plus il est robuste.

Afin d'évaluer de tels modèles, les auteurs de ces contributions ont construit des énumérateurs permettant de parcourir les mots possibles du modèle entraîné. L'évaluation de l'efficacité de ces algorithmes consiste à faire apprendre le modèle sur une partie de la base de données, la partie d'apprentissage, et de générer de nouveaux mots pour compter combien de fois ces mots apparaissent dans la partie restante de la base. Plus un énumérateur retrouve de mots dans la base de données, plus il est jugé efficace. De cette manière, il devient possible de comparer les algorithmes entre eux afin de déterminer celui qui trouve le plus de mots en un nombre imparti de mots générés. Par exemple, si au bout de 10^9 candidats, l'algorithme A trouve 80% de la base et l'algorithme B en trouve 50%, A est meilleur que B en 10^9 essais. Cette manière de mesurer l'efficacité d'un algorithme s'appelle dans la littérature le "number of guesses" (NoG), correspondant au nombre de mots retrouvés de la base de données en fonction du nombre de mots générés (les tentatives).

Le NoG est une bonne manière d'évaluer l'efficacité d'un énumérateur isolé, en variant les paramètres tels que la base de données attaquée. Cependant, certaines contributions ont tendance à se servir du NoG pour comparer les énumérateurs entre eux, afin de montrer qu'un énumérateur est meilleur pour attaquer une base de données particulière. La comparaison entre deux énumérateurs n'est possible que si le temps nécessaire pour générer un mot est le même pour les deux, ce qui n'est généralement pas le cas. Le NoG n'est pas une mesure capable de prendre en compte d'autres facteurs importants lorsque l'on attaque une base de données de mots de passe en pratique. Les paramètres fondamentaux à prendre en compte lorsque l'on attaque une base de données sont : le temps pour générer un ou plusieurs candidats, la mémoire utilisée par le processus, la capacité de l'énumérateur à être parallélisé. Ces paramètres fondamentaux sont essentiels pour choisir la stratégie à adopter afin d'attaquer une base de données. Généralement, c'est le temps de génération des candidats qui est le paramètre le plus important, car c'est celui qui va déterminer le temps d'exécution de l'attaque, tandis que les attaquants ont bien souvent accès à une très grande quantité de mémoire vive leur permettant de ne pas avoir à se

soucier de l'espace mémoire utilisé par le processus. Puisque ces énumérateurs sont utilisés en pratique pour attaquer des bases de données, la prise en compte de ces paramètres fondamentaux est nécessaire à la bonne évaluation des énumérateurs, et par conséquent à la mesure de la performance des attaques les utilisant. Une première piste de travail est donc d'être capable de mesurer la performance des énumérateurs en prenant en compte ces paramètres fondamentaux.

Les services en ligne proposent beaucoup de métriques de robustesse des mots de passe. Les plus répandues sont les métriques heuristiques : l'estimation de la robustesse n'est effectuée qu'en regardant la composition du mot de passe (sa taille et les classes de caractères utilisées). En général, les métriques heuristiques ne sont efficaces que contre la force brute. Plus récemment, des métriques issues de la recherche académique ont été proposées, permettant de prendre en compte des attaques plus avancées que celles considérées par les heuristiques. Une deuxième piste de travail est de se demander à quel point ces métriques de robustesse, qu'elles soient heuristiques ou non, sont efficaces pour se protéger des attaques utilisant les énumérateurs probabilistes. Si ces métriques ne s'avèrent pas suffisamment efficaces, alors cela justifie le fait qu'il faille développer de nouvelles métriques de robustesse qui prennent en compte ces nouvelles attaques.

Afin de proposer des métriques de robustesse qui soient pertinentes pour se protéger des énumérateurs probabilistes, il faut étudier dans quelles conditions ces énumérateurs sont moins performants. Sur l'ensemble des algorithmes de génération de candidats de la littérature, un seul fait l'objet d'une analyse du coût en temps et en espace de son exécution. Tous les autres algorithmes sont étudiés, dans la littérature, en modèle boîte noire, c'est-à-dire que seules les entrées et les sorties sont observées durant son exécution. Une troisième piste de travail est donc d'étudier ces algorithmes en modèle boîte blanche, c'est-à-dire en analysant les structures de données utilisées afin de comprendre comment elles évoluent au cours de l'exécution, notamment pour repérer les conditions dans lesquelles ces énumérateurs voient leur performance baisser. Comprendre la manière dont évoluent leurs structures de données permet de fixer des objectifs plus précis que doivent atteindre les mesures de robustesse pour rendre plus coûteuse en temps et en mémoire la génération des candidats.

Contributions

Le premier travail nous a permis de proposer une formalisation de la performance qui tient compte de paramètres importants que sont la vitesse d'énumération des candidats, la vitesse de la fonction de hachage et le taux de succès de l'algorithme d'énumération. Puisqu'il n'est pas possible d'effectuer les mesures exactes nécessaires au calcul de cette performance de manière précise en un temps raisonnable, nous avons dû mettre en place des techniques pour estimer ces valeurs. Une fois ces estimations effectuées, nous avons montré que même les attaques les plus naïves telle que la force brute sont efficaces dans certaines conditions, par exemple lorsqu'une fonction de hachage rapide est utilisée. Nous avons également montré que la lenteur d'un énumérateur n'est pas un problème lorsqu'une fonction de hachage lente est utilisée, à condition que son taux de succès compense sa lenteur. Ce travail permet

de montrer que si un énumérateur a pour but de casser des mots de passe, alors il doit être évalué avec la bonne méthode, c'est-à-dire en tenant compte également de sa vitesse d'énumération et la vitesse de la fonction de hachage.

En filtrant les bases de données de mots de passe avec les politiques heuristiques étudiées dans le second travail, nous avons conduit des expériences afin de montrer à quel point ces politiques ne sont pas efficaces pour se prémunir de ces attaques. En utilisant les outils formels et techniques présentés dans le premier travail, nous avons montré que même une politique utilisant ZXCVBN, mesure de robustesse présentée comme la référence, avec un score requis maximal, ne permet pas de se prémunir suffisamment des attaques utilisant les modèles probabilistes. Les résultats confirment le fait qu'il faille concevoir de nouvelles métriques de robustesse des mots de passe qui tiennent compte de ces modèles probabilistes, car si l'on peut espérer une augmentation de l'utilisation des fonctions de hachages lentes dans les services en ligne, on constatera également une augmentation de l'usage de ces modèles pour en attaquer les mots de passe.

L'analyse du comportement de deux énumérateurs probabilistes, OMEN et PCFG, d'un point de vue algorithmique dans le troisième travail nous a permis d'observer l'évolution des structures de données utilisées dans ces algorithmes. Ainsi, nous avons été capables de mesurer l'évolution du coût de génération des candidats durant l'énumération. Ce travail aborde le problème de l'analyse des performance d'un énumérateur d'un autre angle. En effet, dans la littérature tout comme dans les deux premiers travaux, les énumérateurs sont généralement étudiés en modèle boîte noire, c'est-à-dire que seules les entrées et les sorties sont étudiées. Ici, nous regardons l'intérieur des algorithmes, apportant des informations supplémentaires permettant de concevoir des méthodes pour rendre plus coûteuse la génération de candidats. Nous montrons par exemple, que si l'on veut que le coût de génération des candidats dans PCFG soit linéaire en le nombre de candidats générés, alors il faut que la taille de la file de priorité augmente de manière exponentielle, ce qui semble difficile à réaliser en pratique.

Enfin, nous rassemblons les conclusions de chaque travail de ce manuscrit pour proposer des pistes de conception des métriques de robustesse modernes, pour lesquelles nous définissons les propriétés que nous jugeons pertinentes à posséder. Même si de prime abord il peut sembler facile de proposer une métrique possédant les deux premières propriétés essentielles que sont la multimodalité et l'adaptabilité, il est loin d'être évident de trouver une métrique qui soit également utilisable.

Chapitre 1

Présentation du domaine

Dans ce chapitre sont présentés les notions et principes essentiels abordés dans le manuscrit ainsi que les différents outils qui ont pu être utilisés lors de cette thèse. Nous présentons également les grandes familles des modèles probabilistes étudiées dans le domaine de la sécurité des mots de passe.

Sommaire

1.1 Terminologie

1.1.1 Mot de passe

Un mot de passe est une chaîne de caractères qui sert à l'authentification d'un individu auprès d'un service ou d'un appareil. Il est souvent couplé à un nom d'utilisateur unique. Dans cette thèse, les secrets qui peuvent être assimilés à des mots de passe tels que les codes PIN, les schémas de déverrouillage de téléphone, et autres, ne sont pas considérés. Seuls les mots de passe choisis par des utilisateurs lors de l'inscription à un service en ligne sont étudiés. En effet, choisir un code PIN, un schéma de déverrouillage ou autre n'est pas la même expérience que celle qui consiste à choisir un mot de passe (quasi-)librement, et cela introduit des biais qui ne sont pas étudiés dans cette thèse.

1.1.2 Fonction de hachage

Une fonction de hachage est une fonction H définie telle que

$$H : \Sigma^* \to \Sigma^k.$$

C'est une fonction qui associe à une donnée en entrée de taille arbitraire, une donnée en sortie de taille fixe appelée empreinte, permettant d'identifier rapidement la donnée d'entrée. On considère généralement trois propriétés de sécurité pour une fonction de hachage.

Résistance à l'attaque sur la première pré-image : étant donnée l'empreinte $H(m)$, il est difficile de retrouver m.

Résistance à l'attaque sur la seconde pré-image : étant donnés $H(m)$ et m, il est difficile de trouver m' tel que $H(m') = H(m)$.

Résistance aux collisions : il est difficile de trouver deux mots m et m' tels que $H(m) = H(m')$.

Dans le cadre de cette thèse, la résistance à l'attaque sur la seconde pré-image n'intervient pas, puisque cela implique d'avoir une première pré-image, donc de pouvoir déjà s'authentifier. Même si les deux autres propriétés doivent être garanties, il est généralement plus facile quand on s'attaque aux mots de passe de chercher des pré-images correspondantes aux empreintes attaquées, à cause des biais introduits dans la création des mots de passe. De plus, d'autres propriétés de sécurité supplémentaires nécessaires pour le stockage du mot de passe sont présentées dans la section 1.2.2.

1.1.3 Candidat

Lors d'une attaque sur les mots de passe, un candidat est un mot généré par l'attaquant, qu'il compare avec un ou plusieurs mots de passe originaux dans le but de trouver le plus de mots de passe possible. Générer des candidats est une étape nécessaire lorsque les mots de passe ne sont pas directement accessibles, c'est-à-dire protégés par des fonctions de hachage ou de chiffrement. Dans ce cas, la stratégie la plus intéressante pour l'attaquant est de tester des candidats un par un et de vérifier si leurs empreintes correspondent à celles attaquées.

1.1.4 Modèle probabiliste et énumérateur parfait

Un modèle probabiliste de mot de passe est une modélisation de la manière dont sont construits les mots de passe.

Modèle probabiliste : Soit $\mathcal{W} = \{w_1, w_2, w_3, ...\}$ un ensemble de mots (fini ou) dénombrable. On définit un modèle probabiliste $\mathcal{M}$ qui associe pour chaque mot $w \in \mathcal{W}$ la probabilité $p(w)$ qu'un utilisateur choisisse w comme mot de passe. On obtient ainsi la distribution de probabilité p telle que

$$\sum_{w \in \mathcal{W}} p(w) = 1.$$

Énumérateur parfait : $E_{\mathcal{M}}$ est un énumérateur parfait basé sur le modèle $\mathcal{M}$ qui énumère les mots $w_1, w_2, w_3, ...$ tels que $p(w_1) \geq p(w_2) \geq p(w_3)...$ $E_{\mathcal{M}}$ énumère les mots par probabilité décroissante au sens large d'après le modèle.

Rang d'un mot : Soit X une variable aléatoire telle que $X \overset{p}{\leftarrow} \mathcal{W}$, c'est-à-dire que X est choisie dans $\mathcal{W}$ avec la distribution p. On définit la fonction $R : \mathcal{W} \to \{1, ..., |\mathcal{W}|\}$ qui calcule le rang d'un mot $w \in \mathcal{W}$, et on définit à partir de X la variable aléatoire $G = R(w)$ qui donne le nombre de candidats générés par $E_{\mathcal{M}}$ pour trouver w.

Robustesse en moyenne : On note $E[G]$ la valeur moyenne de G telle que

$$E[G] = \sum_{i=1}^{|W|} i.p_i = \sum_{w \in \mathcal{W}} R(w)p(w).$$

$E[G]$ est la robustesse en moyenne d'un mot de $\mathcal{W}$. Cette formule est valable également pour les énumérateurs probabilistes (non-parfaits), où seule la fonction R diffère car les candidats ne sont plus énumérés dans l'ordre défini par $\mathcal{M}$.

Attaque d'une base de données $\mathcal{D}$: On utilise E pour attaquer une base de données $\mathcal{D}$ de mots de passe en essayant les mots dans l'ordre d'énumération de E. On définit la performance $P(E, c)$ de E sur $\mathcal{D}$ après c candidats énumérés comme le rapport du nombre de mots trouvés par E après les c essais divisé par le cardinal de $\mathcal{D}$, c'est-à-dire :

$$P(E, c) = \frac{|w \in \mathcal{D}, R(w) \leq c|}{|\mathcal{D}|}.$$

Pour comparer deux modèles $\mathcal{M}_1$ et $\mathcal{M}_2$, on fixe $0 < t \leq 1$ un ratio de $\mathcal{D}$, et on cherche pour chaque énumérateur parfait $E_{\mathcal{M}_1}$ et $E_{\mathcal{M}_2}$, les plus petit c_t^1 et c_t^2 tels que $P(E_{\mathcal{M}_1}, c_t^1) = t$ et $P(E_{\mathcal{M}_2}, c_t^2) = t$. On dit que $\mathcal{M}_1$ est meilleur que $\mathcal{M}_2$ pour une t-attaque si $c_t^1 < c_t^2$.

Exemple : Soit $t = \frac{1}{2}$. Soient $P(E_1, c_t^1 = 10^6) = t$ et $P(E_2, c_t^2 = 10^9) = t$. E_1 trouve la moitié de $\mathcal{D}$ en moins de tentatives que E_2. On peut donc en conclure que $\mathcal{M}_1$ est meilleur pour une $\frac{1}{2}$-attaque que $\mathcal{M}_2$.

Les grandes familles de modèles probabilistes sont présentées dans la section 1.5.

1.1.5 Énumérateur probabiliste

En pratique, énumérer les candidats par probabilité décroissante est très coûteux, car les mots de probabilités proches n'ont pas de raison de se ressembler. Par exemple, “password” et “123456” sont deux mots de passe très probables (parmi les plus utilisés) mais pourtant structurellement très éloignés. Si un énumérateur devait les générer à la suite, il lui faudrait effectuer davantage d'opérations que pour deux mots proches. Afin d'être efficace dans la génération des candidats, il est nécessaire de gérer un compromis entre le coût de génération des candidats et leur production dans un ordre décroissant de probabilité. Un énumérateur probabiliste $E_{\mathcal{M}}$ est donc un algorithme efficace de génération de candidats basé sur un modèle $\mathcal{M}$. $E_{\mathcal{M}}$ n'est pas parfait car il ne respecte pas la condition $p(w_1) \geq p(w_2) \geq p(w_3)...\forall w \in \mathcal{W}$. Cependant,

l'objectif des énumérateurs probabilistes est le suivant : si l'on regroupe les candidats générés par paquets W_i de N candidats, et que l'on note

$$\Pi(W_i) = \sum_{w \in \mathcal{W}} \frac{p(w)}{N}$$

la probabilité moyenne des mots de W_i, alors on souhaite que

$$\Pi(W_i) > \Pi(W_{i+1}),$$

c'est-à-dire que les probabilités moyennes des mots des W_i décroissent au long de l'énumération. Les énumérateurs, à défaut de pouvoir générer les mots par probabilité décroissante, parcourent des ensembles de mots pour lesquelles la probabilité moyenne décroît. Ce compromis reste cohérent avec la t-attaque définie précédemment car après un grand nombre de candidats, l'ordre dans lequel ils ont été générés a peu d'influence sur la performance.

Il est toujours possible de comparer ces énumérateurs entre eux avec le calcul de $P(E, c)$. Cependant, dire que $c_t^1 < c_t^2$ suffit à dire que E_1 est meilleur que E_2 pour une t-attaque, mais ne permet plus de dire que $\mathcal{M}_1$ est meilleur que $\mathcal{M}_2$.

Ces énumérateurs sont dits probabilistes du fait qu'ils se reposent sur un modèle probabiliste, même si tous les algorithmes qui le composent sont déterministes. Ils fonctionnent en deux étapes : l'apprentissage et l'énumération.

Apprentissage : cette étape consiste à entraîner le modèle probabiliste sur des données réelles, lui permettant d'avoir une estimation de la loi de distribution des mots de passe. Le choix des données d'entraînement est donc très important, car il faut à la fois proposer suffisamment de données pour avoir une bonne estimation de la distribution, et à la fois proposer des données les moins biaisées possibles. Un des risques souvent relevé dans le domaine de l'apprentissage statistique est le phénomène de sur-apprentissage : lorsque l'on fournit trop de données sur lesquelles s'entraîner, la modèle fini par en apprendre les caractéristiques spécifiques.

Énumération : cette deuxième étape consiste à énumérer les mots candidats en suivant une stratégie correspondante au choix du compromis entre énumérer les mots par ordre de probabilité selon le modèle et le coût pour les générer. Cela nécessite de maintenir une structure de données à parcourir, définie selon le modèle probabiliste et la stratégie adoptée pour le compromis.

Évaluation : pour évaluer ces énumérateurs, il faut donc choisir une base de données d'entraînement et une base de données d'évaluation. Pour cela, on utilise généralement une base de mots de passe publique $\mathcal{D}$. On effectue une partition $(\mathcal{D}_a, \mathcal{D}_e)$ de $\mathcal{D}$ où $\mathcal{D}_a$ sert à l'apprentissage et $\mathcal{D}_e$ sert à l'évaluation de l'énumération. Après avoir entraîné le modèle sur $\mathcal{D}_a$, on calcule $P(E_{\mathcal{M}}, c)$ en utilisant $\mathcal{D}_e$. Dans les travaux réalisés durant cette thèse et sauf indication contraire, on partitionne $\mathcal{D}$ aléatoirement en deux ensembles de même taille.

1.1.6 Base de mots de passe

Une base de mots de passe est un fichier, souvent sous format texte, composé d'un ensemble de mots de passe. Ces mots de passe sont généralement organisés

sous forme de liste, où chaque élément correspond à un compte utilisateur. La plupart du temps, quand une base de mots de passe fuite, elle contient également des informations personnelles supplémentaires au mot de passe, comme l'identifiant du compte, l'adresse mél, le nom, le prénom, ou l'adresse de l'internaute.

Juridiquement, détenir une base de données avec les informations personnelles des utilisateurs relève du recel, car ces bases de données sont issues d'un crime ou d'un délit. Cependant, dans les travaux de cette thèse, les informations personnelles autre que les mots de passe ne sont pas utilisées, uniquement les mots de passe ont donc été conservés. Aussi, le fait de ne conserver que les mots de passe ne permet plus de leur associer une identité.

La robustesse des mots de passe dépend du service sur lequel ils sont utilisés. Un utilisateur ne porte pas autant d'importance à un compte LinkedIn ou Myspace qu'à son compte Facebook ou son compte mail.

Les bases de données utilisées durant cette thèse sont maintenant présentées.

Rockyou est la base publique la plus utilisée dans la recherche, car les mots de passe sont en clair et elle est facilement trouvable sur Internet. Rockyou est une entreprise proposant des services de réseau social et de jeu vidéo. En 2009, une vulnérabilité SQL a permis à l'attaquant de récupérer les données d'authentification des 32 millions de compte. La politique de composition de mots de passe interdisait d'utiliser des caractères spéciaux. L'entreprise a mis plusieurs jours avant de notifier aux utilisateurs que leurs données avaient fuité.

LinkedIn est un réseau social proposant de mettre en lien les professionnels pour afficher leur carrière, leur emploi, leurs offres d'emplois, ... En 2012, un attaquant se procure la base de données contenant 164 millions d'adresses mail et de mots de passe. Les données n'étaient pas disponibles jusqu'en 2016 où elles ont été vendues sur le marché noir. Les mots de passe étaient hachés avec SHA-1 sans sel, ce qui a permis d'en casser une très grande partie en quelques jours. Aujourd'hui, 98% des mots de passe ont été retrouvés [44].

Myspace est un réseau social permettant d'écrire un blog et de partager ses compositions musicales. En 2008, 360 millions d'adresses mail, de noms d'utilisateur et de mots de passe sont récupérés par un attaquant. Ces données sont vendues en 2016 sur le marché noir. Les mots de passe étaient mis tout en minuscules, tronqués à 10 caractères et hachés avec SHA-1 sans sel. Aujourd'hui, 98% des mots de passe ont été retrouvés [44].

Troy Hunt est un expert en sécurité informatique, propriétaire du site "Have I Been Pwned ? (HIBP)". Ce site web permet à un utilisateur de vérifier si ses données personnelles ont été compromises après une fuite de base de données. En 2017, Troy Hunt rend disponible au téléchargement 300 millions de mots de passe protégés avec SHA-1, sans sel. Ces mots de passe résultent de l'agrégation de plusieurs bases de données ayant fuités. Aujourd'hui, 99,69% des mots de passe ont été retrouvés [44], dont une grande proportion étaient déjà cassés car faisant partie de précédentes fuites.

1.2 Stockage des mots de passe

1.2.1 Généralités

Lorsque l'on souhaite s'authentifier auprès d'un service sur internet, dans la grande majorité des cas c'est la connaissance d'un secret qui est retenue comme méthode d'authentification. Il est donc nécessaire au service de retenir l'ensemble des secrets correspondants à chacun de ses utilisateurs.

Il est évident que les secrets ne doivent pas être stockés tels quels, en clair, puisqu'une intrusion serait suffisante à l'attaquant pour prendre connaissance des mots de passe de tous les utilisateurs. Aussi, il serait possible pour les administrateurs des systèmes de connaître les mots de passe des utilisateurs, même si ce n'est pas leur seul moyen de les connaître. Pour remédier à ce problème, les fonctions de hachage sont souvent utilisées. Lors de l'inscription, l'empreinte du mot de passe est stockée dans la base de données, et lors de la connexion, l'empreinte est recalculée puis comparée à celle enregistrée lors de l'inscription. Puisqu'on estime qu'une collision est peu probable, une correspondance entre les empreintes permet d'accepter la connexion de l'utilisateur. Le chiffrement des mots de passe par une fonction cryptographique symétrique fait débat, car un attaquant qui retrouverait la clé de chiffrement pourrait déchiffrer tous les mots de passe, même si elle n'est généralement pas stockée au même endroit que la base de mots de passe. Par exemple, la base de données d'Adobe a fuité en 2013, contenant 153 millions de comptes avec nom d'utilisateur, adresse mail, mot de passe chiffré avec 3DES et un indice sur ce mot de passe (en cas d'oubli). La clé de chiffrement n'a toujours pas été retrouvée, même si les indices sur les mots de passe permettent d'avoir une bonne confiance sur les mots de passe correspondants. Il existe d'ailleurs des mots croisés à ce sujet [14].

Un second problème qui se pose est le fait que deux utilisateurs partageant le même mot de passe auraient la même empreinte dans la base de données, ce qui n'est pas souhaitable dans le cas d'une attaque car cela facilite le travail de l'attaquant. Pour résoudre ce problème est introduite la notion de *sel* : c'est une chaîne aléatoire qui préfixe le mot de passe avant de calculer son empreinte et qui est stockée à ses côtés dans la base de données. Ainsi, même si deux utilisateurs partagent le même mot de passe et à condition que la taille du sel soit suffisamment grande, ils ne partagerons pas la même empreinte dans la base. Ceci oblige l'attaquant à recalculer les empreintes avec tous les sels de la base pour chacune des tentatives qu'il effectue.

1.2.2 Choix de la fonction de hachage

Il existe beaucoup de fonctions de hachage disponibles dans la littérature. Cependant, les besoins en sécurité pour stocker les mots de passe et en cryptographie ne sont pas les mêmes. En général, la taille des entrées en cryptographie est grande, ce qui n'est pas le cas des mots de passe, qui dans la grande majorité des cas sont très courts. Du fait que les mots de passe soient des entrées courtes, mais également du fait que ce soient des humains qui les génèrent, l'attaquant a un avantage à essayer des mots candidats puis à comparer leurs empreintes à celles attaquées.

Les fonctions de hachage cryptographiques telles que SHA-1, SHA-256, SHA-512, SHA3, MD5, ... sont conçues pour être rapides, résistantes aux collisions, et aux attaques par pré-image. Dans un contexte de stockage des mots de passe, la rapidité de la fonction de hachage donne un avantage à l'attaquant, car il devient capable d'essayer beaucoup de candidats dans un temps imparti. Une fonction de hachage particulière, celle utilisée dans Windows (jusqu'à Windows 7 pour compatibilité), est NTLM. Elle avait la particularité de mettre le mot de passe en majuscules et de hacher séparément ses 7 premiers octets et ses 7 derniers, ce qui donnait un avantage conséquent à l'attaquant, car il est plus facile d'attaquer deux mots de passe de 7 octets qu'un seul de 14 octets.

Des fonctions de hachage ont donc été développées afin de pouvoir stocker les mots de passe de manière plus sécurisée. On peut citer bcrypt [43], scrypt [42], PBKDF2 [32] et Argon2 [2]. Ces fonctions ont l'avantage d'être lentes, c'est-à-dire que calculer l'empreinte d'un mot requiert de répéter une opération coûteuse un grand nombre de fois. Par exemple, bcrypt repose sur la répétition de la fonction d'initialisation des clés de la fonction de chiffrement Blowfish, qui est coûteuse. Le deuxième avantage de ces fonctions, c'est qu'elles sont adaptatives, c'est-à-dire qu'il est possible de paramétrer le coût nécessaire (en temps, en espace mémoire, en nombre de threads) pour calculer une empreinte. Cela consiste simplement à faire varier le nombre de fois où l'opération coûteuse est répétée. De cette manière, il devient possible de s'adapter à l'augmentation de la puissance de calcul de l'attaquant. Ces deux avantages permettent aussi de rendre beaucoup moins intéressant l'utilisation des cartes graphiques et des circuits imprimés dédiés au cassage des mots de passe (ASIC, FPGA), car il ne devient plus possible de grandement paralléliser les opérations dû au fait que ces fonctions sont coûteuses en mémoire. Les ASIC et les FPGA sont devenus très populaires (et donc plus abordables) avec l'arrivée du minage des cryptomonnaies.

Il est donc aujourd'hui vivement recommandé d'utiliser ces fonctions de hachage lentes et coûteuses en mémoire. Elles ont tendance à être de plus en plus utilisées, même si elle ne sont pas encore majoritaires dans les fuites de donnée que l'on constate ces dernières années. La figure 1.1 montre la distribution des fonctions de hachage utilisées dans les fuites récentes. Ces statistiques sont cependant sûrement biaisées, du fait qu'un service qui utilise une mauvaise fonction de hachage a probablement plus de risques de se faire voler sa base à cause de mauvaises pratiques de sécurité.

Ces fonctions ont tout de même le désavantage d'augmenter la charge de l'infrastructure du service qui les utilise, mais la sécurité a un prix. Ce désavantage n'est cependant rien comparé à la sécurité offerte par ces fonctions.

Quelque soit le choix de la fonction et de ses paramètres, l'utilisation d'une fonction de hachage lente est déjà une progression majeure comparée à l'utilisation des fonctions rapides. Si l'on en vient à se poser la question de quelle fonction lente choisir et avec quels paramètres, c'est déjà qu'une bataille a été gagnée. Cependant, il est raisonnable de ne pas choisir des paramètres plus petits que ceux par défaut. Si l'on se réfère à la publication spéciale du NIST [38], le nombre d'itération pour PBKDF2 doit être d'au moins 10000, ce qui équivaut à un coût 10 pour bcrypt (2^{10} itérations) et 512 tours pour Argon2i (avec $m = 8Kb$, $p = 1$) sur un ordinateur

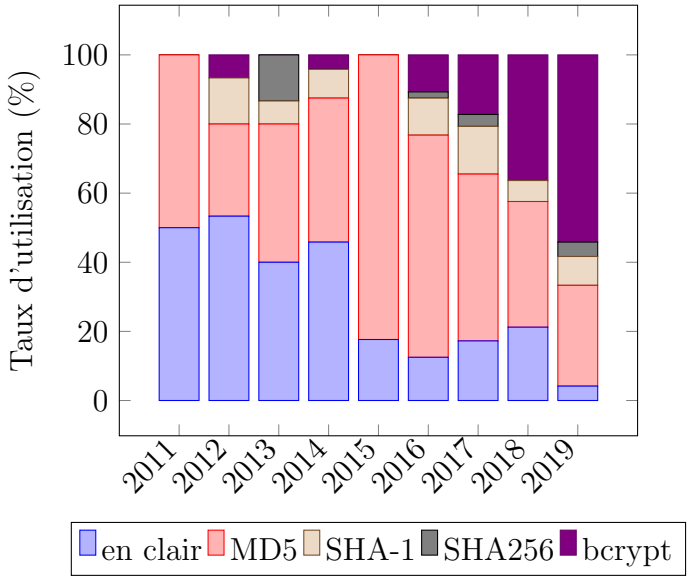


FIGURE 1.1 – Distribution de l'utilisation des fonctions de hachage dans les fuites de mots de passe depuis 2011. Source : haveibeenpwned.com

portable moyen.

1.3 Attaques

Il existe différentes attaques sur les mots de passe, elles sont présentées ici. Chaque attaque (sauf quelques cas) peut être utilisée soit pour une attaque en ligne ou hors-ligne.

1.3.1 Types d'attaque

Il existe deux familles d'attaques sur les mots de passe : en ligne et hors-ligne. Lorsque l'on attaque des mots de passe, on essaie de deviner les mots de passe un à un en effectuant une requête auprès d'un oracle afin de savoir si la tentative est correcte. Ces deux familles d'attaques permettent de distinguer deux oracles :

— attaque en ligne : l'oracle est à distance, sur le réseau, détenu par le serveur et hors de contrôle de l'attaquant

— attaque hors-ligne : l'oracle est local, hors du réseau, détenu par l'attaquant et partiellement sous le contrôle de l'attaquant

1.3.1.1 En ligne

Une attaque en ligne consiste pour l'adversaire à se rendre sur l'interface de connexion du service, jouant le rôle d'oracle, et à essayer différentes combinaisons nom d'utilisateur / mot de passe. Cette attaque requiert une connexion au réseau et demande de questionner le service à chaque tentative. Elle est par conséquent très coûteuse pour l'attaquant.

Cette attaque est composée de deux étapes pour chaque tentative : générer un candidat, et vérifier auprès du service en ligne s'il est correct. L'oracle peut être configuré pour accepter un nombre de requêtes limité.

1.3.1.2 Hors-ligne

Contrairement à l'attaque en ligne, l'attaque hors-ligne ne requiert pas de connexion au réseau. Après s'être procuré une copie de la base de données contenant la liste mots de passe protégés du service, l'attaquant se charge lui-même de vérifier si les couples nom d'utilisateur / mot de passe sont présents. L'attaquant joue lui-même le rôle de l'oracle. Cette attaque requiert cependant une intrusion (par l'attaquant ou une entité tierce) dans le système du service en question pour y récupérer la base de données. Le coût de l'attaque dans ce cas dépend du niveau de protection offert par le service sur les mots de passe ainsi que des ressources matérielles dont dispose l'attaquant.

Cette attaque se décompose en trois étapes : générer un candidat, le hacher, et vérifier s'il est dans la base de données. Le nombre de requêtes auprès de l'oracle est illimité, mais des contre-mesures comme les fonctions de hachage lentes permettent de rendre coûteuses les requêtes.

Dans ce manuscrit, on se place dans un scénario d'attaque hors-ligne, et on ne considère pas les attaques en ligne.

1.3.2 Force brute

Probablement l'attaque la plus connue du grand public, la force brute est un énumérateur non-probabiliste. Elle consiste à tester toutes les combinaisons possibles d'un jeu de caractères donné, de manière incrémentale. Cette attaque garantit de trouver les mots de passe attaqués, à condition que les caractères qui les composent soient dans le jeu de caractères utilisé. Cependant, elle requiert un temps infini pour garantir de trouver tous les mots, c'est pourquoi en pratique ces attaques sont tronquées. En général, si une fonction de hachage rapide est utilisée pour protéger les mots de passe, c'est la première attaque lancée. Si l'on considère un jeu de caractères de taille 95 (26 lettres minuscules, 26 majuscules, 10 chiffres, 33 caractères spéciaux), il faut 3h45 et 8 cartes graphiques GTX 1080Ti pour casser tous les mots de passe de moins de 8 caractères protégés avec NTLM [19].

Calcul du rang d'un mot m pour la force brute :

Soit l la taille du mot $m = c_1...c_l$, et soit $\mathcal{A}$ l'alphabet utilisé par la force brute. Le rang $R(m)$ est calculé tel que

$$R(m) = \mathcal{A}_{c_l} - 1 + \sum_{i=1}^{l-1} |\mathcal{A}|^i + (\mathcal{A}_{c_i} - 1)|\mathcal{A}|^{l-i}$$

où $\mathcal{A}_{c_i}$ est la position de la lettre c_i dans l'alphabet $\mathcal{A}$.

preuve : énumérer $m = c_1...c_l$ peut se décomposer en 2 étapes : 1) énumérer tous les mots de taille $1 \leq i < l$, puis 2) énumérer les mots de taille l qui précèdent m dans l'ordre lexicographique.

1) Il y a $\sum_{i=1}^{l-1} |\mathcal{A}|^i$ mots de taille $1 \leq i < l$.
2) pour compter les mots de taille l qui précèdent $m = c_1...c_l$ dans l'ordre lexicographique, il faut compter les mots $w = w_1...w_l$ tels que $w_1 < c_1$, puis les mots $w = c_1w_2...w_l$ tels que $w_2 < c_2$, ainsi de suite jusqu'aux mots $w = c_1...c_{l-1}w_l$ tels que $w_l < c_l$. Ceci est équivalent, pour chaque lettre c_i du mot m, à énumérer $\mathcal{A}_{c_i}$ fois tous les mots de taille $l - i$. Il y a donc

$$\sum_{i=1}^{l} (\mathcal{A}_{c_i} - 1)|\mathcal{A}|^{l-i}$$

mots de taille l qui précèdent m dans l'ordre lexicographique. On a donc

$$\begin{aligned} R(m) &= \sum_{i=1}^{l-1} |\mathcal{A}|^i + \sum_{i=1}^{l} (\mathcal{A}_{c_i} - 1)|\mathcal{A}|^{l-i} \\ &= \mathcal{A}_{c_l} - 1 + \sum_{i=1}^{l-1} |\mathcal{A}|^i + (\mathcal{A}_{c_i} - 1)|\mathcal{A}|^{l-i} \end{aligned}$$

1.3.3 Dictionnaire

Un dictionnaire est une liste de mots qui peuvent à la fois provenir du dictionnaire d'une langue ou bien être une compilation de mots de passe précédemment cassés. Une attaque par dictionnaire consiste donc à choisir un dictionnaire et à essayer chacun de ses mots un par un. Elle est bien souvent utilisée dans un scénario d'attaque hors-ligne. Cependant dans le cas d'une attaque en ligne, un dictionnaire bien plus petit et personnalisé est plus intéressant compte tenu des protections réseau possibles. Cette attaque est très rapide car elle ne requiert aucun calcul pour générer les candidats. Elles est également l'une des premières attaques lancées lorsque la fonction de hachage est rapide.

1.3.4 Règles de réécriture

Cette technique n'est pas une attaque au sens propre mais se combine à une attaque, le plus souvent celle par dictionnaire. Elle consiste à transformer les candidats en appliquant des règles de remplacement, d'ajout, de suppression, d'inversion de caractères afin d'imiter les méthodes utilisées par les humains pour créer leurs mots de passe.

Une règle de réécriture $R : A \to X$ est une fonction qui remplace toutes les occurrences du mot A par le mot X dans un mot w. Les règles de réécriture s'appliquent donc à des mots et permettent de produire d'autres mots. Par exemple, si $w = abaaaababb$ et $R : a \to c$, alors $R(w) = cbccccbcbb$. L'exemple le plus connu est le leetspeak, où l'on remplace des lettres par des chiffres ou des caractères spéciaux, par exemple le mot "password" devient "p@$$w0rd". Si $\mathcal{W}$ est l'ensemble des mots de départ, et $\{R_1, ..., R_n\}$ un ensemble de règles de réécriture différentes, alors l'ensemble des mots produits correspond à l'application de chaque règle sur chaque mot de $\mathcal{W}$ (il peut y avoir des doublons) et son cardinal est $N = n \times Card(\mathcal{W})$. De ce fait, N peut être très grand si l'on utilise un grand dictionnaire avec beaucoup de règles. Par exemple, le dictionnaire "openwall" de John The Ripper, contient 5 millions de mots, et Hashcat propose une compilation des 64 règles de réécriture les plus utilisées.

Si l'on combine les deux, on obtient un ensemble de 320 millions de mots. C'est pourquoi il est nécessaire, pour un attaquant, de limiter la taille du dictionnaire ou le nombre de règles en fonction de sa puissance de calcul.

1.3.5 Compromis temps-mémoire

Les compromis temps-mémoire, dans le cadre des attaques sur les mots de passe, permettent d'effectuer des pré-calculs sur les fonctions de hachage utilisées. La première publication d'un tel compromis est celle de Martin Hellman [20] qui est destinée à casser des clés cryptographiques. Dans cette publication, Hellman propose une manière efficace de stocker des pré-calculs de pré-images d'une fonction de hachage. Pour cela, on calcule des chaînes en appliquant successivement à un mot de départ m la fonction de hachage H et une fonction de réduction R. R prend une empreinte, sortie de H, en entrée et produit un mot dans l'espace que l'on veut couvrir. On calcule ainsi $e = H(R(H(R(...R(H(m))))))$ et on stock m et e. Il est possible ainsi de savoir si l'empreinte $H(m')$ d'un mot m' que l'on souhaite retrouver est dans la table. Soit $e' = H(m')$. Si e' est stockée dans la table à la chaîne i, alors $m' = R(H(R(...R(H(m_i)))))$, c'est-à-dire que l'on retrouve m' en appliquant toute la chaîne sauf la dernière fonction de hachage. Si e' n'est pas dans la table, alors on calcule $e'' = H(R(e'))$ et on cherche si e'' est dans la table, de la même manière qu'avec e' (sauf qu'on effectue un hachage et une réduction de moins que pour retrouver m' quand e' est dans la table). On répète cette opération jusqu'à trouver $e^{(n)}$ qui soit dans la table ou que n dépasse la taille des chaînes de la table.

Le principal problème de cette structure réside dans la fait que R peut provoquer des collisions (H aussi mais dans une moindre mesure), c'est-à-dire deux empreintes différentes qui produisent le même mot. On dit que les chaînes fusionnent et dans ce cas, la fin d'une chaîne peut avoir plusieurs débuts.

Philippe Oechslin [39] propose de diversifier la fonction R afin de réduire le nombre de collisions et donc de fusions des chaînes (voir Figure 1.2). Ainsi, il fixe le nombre de fonctions de réduction et paramètre chacune d'entre elles avec un entier différent. Même si deux empreintes e_i et e_j produisent le même mot m_{i+1} par des

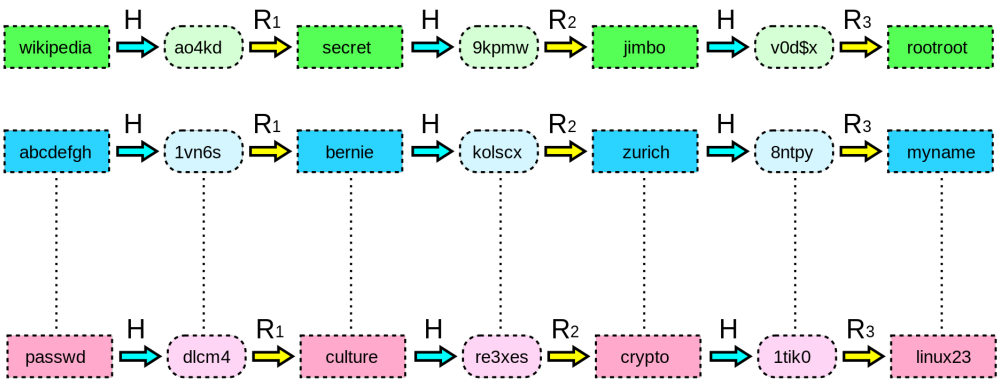


FIGURE 1.2 – Exemple de calcul de table arc-en-ciel (Wikipédia)

fonctions de réductions R_i et R_j, la suite de la chaîne sera différente à condition que $i \neq j$ et donc les chaînes de fusionnent pas.

Les tables arc-en-ciel sont un compromis temps-mémoire très intéressant pour casser des mots de passe. Par exemple, à l'époque de la publication, la fonction de hachage de Windows tronquait les mots de passe à 14 caractères, les mettait en majuscules et hachait séparément la première et la deuxième moitié. Il était possible d'essayer les 2^{37} mots de passe différents (7 caractères parmi 36 avec répétitions) en seulement 13 secondes avec une table arc-en-ciel de $7Go$. Une manière efficace et facile de se prémunir des compromis temps-mémoire est d'utiliser un sel. Il oblige l'attaquant à pré-calculer une table pour chaque valeur de sel possible, ce qui n'est pas faisable lorsque l'univers des sels est suffisamment grand, d'où l'intérêt de choisir un sel de taille suffisante. Bcrypt utilise un sel de 128 bits, ce qui permet de diversifier la fonction de hachage de 2^{128} manières différentes. Même si le sel est une contre-mesure très efficace face aux tables arc-en-ciel (et aux pré-calculs en général), les fuites de données récentes montrent que son usage n'est pas encore totalement généralisé.

De telles attaques ne sont pas prises en compte dans ce manuscrit.

1.3.6 Filtres de Bloom et dérivés

Lorsque l'on attaque les mots de passe dans un scénario hors-ligne, il est nécessaire de vérifier si les candidats générés sont présents dans la base de données. Cependant, il est impensable de stocker la base de données telle quelle en mémoire, car cette base peut parfois atteindre des tailles allant jusqu'à plusieurs gigaoctets.

Pour résoudre ce problème, on utilise une structure de données probabiliste appelée filtre de Bloom [3]. Cette structure permet de vérifier l'absence d'un élément de manière certaine, mais vérifier la présence d'un élément s'effectue avec une certaine probabilité d'erreur. Il peut donc y avoir des faux positifs mais pas de faux négatifs. L'intérêt d'une telle structure est le fait que sa taille en mémoire soit fixe et petite comparée à la taille de l'ensemble des éléments qu'elle contient.

Ces filtres sont construits de la manière suivante. Soit T un tableau de m bits, initialisé à 0. Soit P un ensemble d'élément à stocker. Soient $h_i, 1 \leq i \leq k$ un ensemble de fonctions de hachage telles que $h_i : P \to [1, m]$.

Insertion : pour insérer un élément $e \in P$, on calcule $t_i = h_i(e)$ et on écrit $T[t_i] = 1$ pour $1 \leq i \leq k$.

Vérification : pour vérifier si un élément $e \in P$ est présent, on calcule $t_i = h_i(e)$ et vérifie si $T[t_i] = 1$ pour $1 \leq i \leq k$. Si au moins une case n'est pas à 1, alors l'élément est absent avec certitude. Si toutes les cases sont à 1, alors l'élément est généralement présent avec forte probabilité.

Il peut y avoir des faux positifs du fait que les $T[t_i]$ ont pu être mis à 1 par plusieurs insertions auparavant. Dans le cas où un élément est annoncé présent, le logiciel de cassage de mot de passe fera une vérification de présence supplémentaire dans la base de données elle-même. Ce système a quand même l'avantage de réduire grandement le nombre de recherches dans la base de données car un candidat est en moyenne plus souvent absent que présent dans le filtre de Bloom.

1.4 Outils

Dans cette section sont présentés les outils utilisés pendant cette thèse.

1.4.1 Hashcat

Hashcat est un logiciel libre de cassage de mots de passe [47]. Il s'utilise en ligne de commande et permet d'utiliser les cartes graphiques de la machine pour casser une liste de mots de passe protégés par une fonction de hachage. Il supporte près de 400 fonctions de hachage différentes. Il possède plusieurs modes d'attaques :

— par dictionnaire avec ou sans règle de réécriture

— utilisant un modèle de chaîne de Markov

— par combinaison de deux dictionnaires (les candidats sont le produit cartésien des deux ensembles, par concaténation)

Une fonctionnalité ajoutée récemment dans Hashcat et qui tisse un pont avec le travail effectué durant cette thèse est la fonction "slow candidates". Il s'agit d'une interface logicielle permettant d'y attacher des énumérateurs de candidats avancés qui pour la plupart sont lents, tels que OMEN, la PCFG de Weir, ... Cela permettra dans le futur d'implanter plus facilement les algorithmes de la littérature au sein même d'Hashcat, évitant ainsi d'avoir à communiquer à travers des tubes entre le programme énumérateur et le programme qui appliquait la fonction de hachage. Le second intérêt majeur d'une telle interface est de pouvoir effectuer le maximum de travail sur le CPU puisque les fonctions de hachage lentes empêchent d'exploiter le parallélisme des cartes graphiques.

Le mode Markov implanté dans Hashcat est une version modifiée du modèle présenté en section 1.5.1 : il tient compte des positions des caractères. Lors de sa phase d'apprentissage, Hashcat entraîne le modèle en mémorisant également la position du n-gram dans le mot. Lors de la phase d'attaque, Hashcat choisit d'abord les chaînes qui satisfont les positions.

De part son usage intensif des cartes graphiques et du langage OpenCL, les phénomènes étudiés dans cette thèse sont parfois difficiles à mesurer. Par exemple, mesurer le temps nécessaire pour générer un candidat n'a pas réellement de sens quand les calculs se font de manière parallèle.

1.4.2 John The Ripper

John The Ripper est un logiciel libre de cassage de mots de passe [25]. Il est à l'origine destiné à récupérer les mots de passe d'une machine UNIX, mais supporte désormais plus d'une centaine de fonctions de hachage. Il est populaire du fait qu'il peut fonctionner sans avoir besoin d'être configuré, même s'il possède des paramètres avancés pour les utilisateurs plus aguerris. John The Ripper utilise essentiellement le CPU et permet nativement de paralléliser et distribuer une attaque sur plusieurs nœuds d'un réseau, même si la manière de répartir le travail est rustique (pas de communication entre les nœuds). Il possède plusieurs modes d'attaques :

— “single” : utilisation des informations personnelles des comptes utilisateurs à attaquer (nom d’utilisateur, ...)

— “incremental” : un mode simple basé sur un modèle de chaîne de Markov d’ordre 2

— “external” : permet à l’attaquant de coder une stratégie de génération des candidats

— “Markov” : utilisation d’un modèle de chaîne de Markov configurable dans les fichiers de configuration

— “wordlist” : attaque par dictionnaire

— “rules” : application des règles de réécriture

Puisque John The Ripper ne fait quasiment pas usage des cartes graphiques pour attaquer les mots de passe, il est plus facile d’y intégrer des mesures utiles dans cette thèse. Par exemple, mesurer le temps pour générer un candidat peut se faire facilement à condition que les calculs soient effectués de manière séquentielle.

Calcul du rang d’un mot m pour le mode Markov de JtR : L’estimation du rang $R(m)$ dans l’énumération de JtR-Markov se fait en calculant un encadrement de ce rang. Après avoir effectué l’apprentissage sur une base de données, on dispose d’un fichier contenant les statistiques calculées par l’algorithme d’apprentissage. Il se présente sous la forme suivante :

```
proba1(97)  = 97
proba2(112) = 51
proba2(122) = 44
proba2(100) = 51
...
```

où chaque ligne contient la fréquence d’un 1-gram ou 2-gram. Pour chaque ligne, la partie droite de l’égalité correspond au niveau du N-gram, c’est-à-dire la valeur logarithmique discrétisée de sa fréquence, ce qui permet de travailler avec des sommes plutôt qu’avec des produits. Plus le niveau est grand, moins le N-gram est fréquent. Ensuite, “proba1” signifie que la fréquence correspond à la première lettre d’un mot (un 1-gram), et “proba2” à la fréquence d’un 2-gram. Entre parenthèses se situent les codes ASCII des lettres composants le N-gram concerné par cette fréquence. La première ligne indique que la lettre ’a’ (97 en ASCII) a un niveau de 97 lorsqu’elle commence un mot, tandis que la deuxième ligne indique que ’ap’ (97 et 122 en ASCII) a un niveau de 51. Le niveau d’un mot m est la somme des niveaux des N-grams qui le composent. Par exemple, le niveau du mot “password” est calculé en ajoutant les niveaux du 1-gram ’p’ et des 2-grams “pa”, “as”, “ss”, “sw”, “wo”, “or”, et “rd”. Ainsi, plusieurs mots peuvent avoir le même niveau. L’encadrement du rang $R(m)$ peut être calculé en utilisant l’outil “genmkvpwd” fourni avec John The Ripper. Étant donné le fichier de statistiques, le niveau n du mot et la taille des candidats à générer, l’outil calcule le nombre de candidats à générer pour atteindre n, ce qui définit la borne inférieure $R_l(m) \leq R(m)$. Il faut ensuite calculer le nombre de candidats à

générer pour atteindre $n+1$, ce qui définit la borne supérieure $R(m) \leq R_s(m)$. On prend alors le milieu

$$\frac{R_l(m) + R_s(m)}{2}$$

de ces bornes comme estimation de $R(m)$.

1.4.3 PACK

PACK (Password Analysis and Cracking Kit) est un ensemble de scripts en Python permettant d'effectuer des analyses statistiques sur une liste de mots de passe. Il est composé de 4 scripts :

— StatsGen : affiche les statistiques relatives à la liste de mots de passe telles que la taille, les caractères utilisés, les masques correspondants ...

— MaskGen : à partir d'une analyse effectuée par StatsGen, MaskGen génère les masques correspondants dans un format utilisable pas les logiciels de cassage de mots de passe

— PolicyGen : permet de générer un ensemble de masques satisfaisant des contraintes imposées aux mots de passe dans un contexte d'entreprise par exemple

— RuleGen : permet de tenter de deviner les règles de réécritures utilisées pour passer d'un mot du dictionnaire à son mot de passe

Puisque PACK est un logiciel relativement lent, nous avons développé notre propre logiciel d'analyse statistique des mots de passe. Les détails techniques sont décrits dans l'annexe A.

1.5 Familles de modèles probabilistes

Dans la littérature, il existe trois grandes familles de modèles probabilistes pour les mots de passe : les chaînes de Markov, les grammaires probabilistes non-contextuelles (PCFG), et les modèles utilisant l'apprentissage profond.

Les modèles probabilistes sont particulièrement inspirés des modélisations effectuées dans le domaine du traitement automatique du langage naturel. Ceci est principalement dû au fait que, bien souvent, les internautes construisent leurs mots de passe en utilisant des mots ou des noms dans leur langue. Ainsi, cela explique pourquoi les attaques basées sur ces modèles fonctionnent particulièrement bien.

1.5.1 Chaînes de Markov

Les chaînes de Markov sont connues pour être efficaces dans la modélisation du langage. Elles permettent d'estimer la probabilité d'apparition d'une lettre de l'alphabet dans un mot en fonction d'un nombre fixe de lettres qui la précèdent.

La probabilité classique $p(m)$ (hors modélisation) d'un mot $m = c_1c_2...c_k$ est

$$p(m) = p(c_1)p(c_2|c_1)p(c_3|c_1c_2)...p(c_k|c_1c_2...c_{k-1}) = \prod_{i=1}^{k} p(c_i|c_1..c_{i-1})$$

c'est-à-dire que la probabilité d'apparition d'une lettre dépend de toutes les précédentes.

Un modèle de chaîne de Markov permet de calculer une estimation ρ_n de cette probabilité avec

$$\rho_n(m) = p(c_1)p(c_2|c_1)...p(c_k|c_{k-n}..c_{k-1}) = \prod_{i=1}^{k} p(c_i|c_{i-n}..c_{i-1})$$

où n, appelé l'ordre, est le paramètre qui détermine la mémoire du processus. Plus n est grand, plus ρ_n est proche de p, mais plus la modélisation est coûteuse en mémoire, car il faut pouvoir stocker tous les n-grams possibles. Dans le cas où $n = 0$, $\rho_0(m) = \prod_{i=1}^{k} p(c_i)$, le processus n'a pas de mémoire.

En 2005, Arvind Narayanan et Vitaly Shmatikov [37] proposent un nouveau compromis temps-mémoire (appelé temps-espace dans la publication) reposant sur un modèle de chaîne de Markov. Le postulat de base est que les utilisateurs ont tendance à reproduire la distribution des lettres de leur langage dans leurs mots de passe. Ainsi, il est possible de réduire l'espace de recherche des mots en exploitant des techniques de modélisation Markoviennes utilisées dans le traitement automatique des langues. Les auteurs proposent dans un second temps un algorithme qui génèrent des candidats en utilisant les pré-calculs effectués. Ceci défini un énumérateur probabiliste basé sur les chaînes de Markov.

L'étape d'apprentissage consiste à calculer les fréquences de tous les n-grams, sous-parties de taille n d'un mot, d'une liste de mots en entrée. L'algorithme génère ensuite, dans sa phase d'énumération, des nouveaux candidats en essayant de les générer du plus probable au moins probable. Dans cet article, les valeurs de n proposées sont 0 et 1 : on parle de chaîne de Markov d'ordre 0 et 1.

Un intérêt supplémentaire de cet algorithme est le fait qu'il dispose d'une fonction d'indexation, qui permet en un temps raisonnable de générer le $i^{éme}$ candidat sans générer les $i - 1$ précédents, ce que ne permettent pas les autres énumérateurs probabilistes d'aujourd'hui.

Cet algorithme est la base des modes Markov des logiciels John The Ripper et Hashcat. OMEN [11] est un énumérateur probabiliste basé sur les chaînes de Markov. Il est présenté en section 3.4.1. OMEN et le mode Markov de John The Ripper sont les deux énumérateurs basés sur les chaînes de Markov étudiés dans ce manuscrit. Une étude plus approfondie de l'algorithme d'OMEN est effectuée dans le chapitre 6.

1.5.2 Grammaires probabilistes non-contextuelles

1.5.2.1 Généralités

Les grammaires probabilistes non-contextuelles (Probabilistic Context-Free Grammars en anglais) sont également efficaces pour modéliser un langage. Une PCFG peut

être définie [29] comme un quintuplet d'ensembles finis $\langle \Sigma, \mathcal{N}, \mathcal{S}, \mathcal{R}, \mathcal{P} \rangle$, où Σ est un ensemble de symboles terminaux, $\mathcal{N}$ un ensemble de non-terminaux, $\mathcal{S}$ le symbole spécial "début" de $\mathcal{N}$, et $\mathcal{R}$ un ensemble de règles de production de la forme $A \to B$ avec $A \in \mathcal{N}$ et $B \in (\Sigma \cup \mathcal{N})^*$. Ces grammaires sont dites non-contextuelles car les parties gauches des règles de production ne sont toujours formées que d'un seul symbole de $\mathcal{N}$. Elles sont dites probabilistes car à chaque règle de production $A \to B$ est associée une probabilité $P_{A \to B} \in [0, 1]$. On a donc une distribution de probabilités sur A notée $P_A(B) = P_{A \to B}$. Le dernier ensemble $\mathcal{P} = \{P_{A \to B} | (A \to B) \in \mathcal{R}\}$ est l'ensemble de toutes les probabilités associées aux règles telles que

$$\forall A \in \mathcal{N}, \sum_{\{B | (A \to B) \in \mathcal{R}\}} P_{A \to B} = 1.$$

On note $\mathcal{W}$ l'ensemble des mots produits par une PCFG. Pour tout mot $w \in \mathcal{W}$, on note R_w l'ensemble des règles qui permettent de le produire. R_w est unique, c'est le seul ensemble qui permet de produire w, car les PCFG sont non-ambiguës. $\mathcal{W}$ est donc un produit cartésien, ce qui nous permet de définir la probabilité

$$p(w) = \prod_{A \to B \in R_w} P_{A \to B}$$

c'est-à-dire que la probabilité de w est le produit des probabilités des règles permettant de le produire.

Dans la littérature, il existe deux énumérateurs utilisant les grammaires probabilistes : l'énumérateur de Weir [54] et celui de Komanduri [29]. Dans cette thèse, seule la grammaire de Weir a été étudiée.

1.5.2.2 Grammaire de Weir

Matt Weir propose en 2009 [54] un algorithme de génération de candidat basé sur des Grammaires Probabilistes Non Contextuelles (PCFG), qu'il fixe comme étant non récursives et non ambiguës. Les candidats ainsi générés correspondent en réalité aux terminaux de la grammaire. Pour cela, Weir définit $Q = (\mathcal{N} \backslash \mathcal{S})$ comme l'ensemble des symboles non-terminaux sans le symbole "début", et les règles de production de la grammaire sont soit de la forme

$$\mathcal{S} \to \mathcal{Q}, \mathcal{Q} \in Q^*$$

soit de la forme

$$A \to T, A \in Q, T \in \Sigma.$$

Les éléments de Q^* sont appelés bases composées. Grâce à ces deux formes de règle, les grammaires ne sont pas récursives car tous les symboles, hormis $\mathcal{S}$, produisent des terminaux. Deux contraintes sont ajoutées à ces grammaires :

— les symboles à gauche et à droite dans une règle doivent être de même longueur

— un terminal ne peut comporter les lettres que d'un seul alphabet

$$
\begin{aligned}
\Sigma &: \{password, baseball, 123, !\} && \text{(terminaux)}\\
\mathcal{N} &: \{\mathcal{S}, L_8, D_3, S_1\} && \text{(non-terminaux)}\\
\mathcal{R} &: \mathcal{S} \to L_8 S_1 && \text{(bases composées)}\\
&\quad \mathcal{S} \to L_8 D_3 S_1 &&\\
&\quad L_8 \to password && \text{(production des terminaux)}\\
&\quad L_8 \to baseball &&\\
&\quad D_3 \to 123 &&\\
&\quad S_1 \to ! &&\\
\mathcal{P} &: P_{\mathcal{S}\to L_8 S_1} = 0.75 && \text{(probabilités)}\\
&\quad P_{\mathcal{S}\to L_8 D_3 S_1} = 0.25 &&\\
&\quad P_{L_8 \to password} = 0.5 &&\\
&\quad P_{L_8 \to baseball} = 0.5 &&\\
&\quad P_{D_3 \to 123} = 1.0 &&\\
&\quad P_{S_1 \to !} = 1.0 &&
\end{aligned}
$$

FIGURE 1.3 – Un exemple de grammaire probabiliste non-contextuelle

Les alphabets définis dans le cadre de ces grammaires sont L, D, S où L contient tous les lettres alphabétiques de la langue anglaise, D les chiffres et S les caractères spéciaux. Ainsi, Q peut s'écrire de la forme $Q = \{L_i, D_i, S_i\}, \forall i \in [1, M]$ où M est une longueur maximale arbitraire des symboles. L'indice i indique la longueur des symboles des parties droites des règles de production (à cause de la première contrainte).

La figure 1.3 montre un exemple d'une PCFG déduite depuis le corpus de mots {*password !*, *password !*, *baseball !*, *baseball123 !*}, où $P_{A\to B}$ est la probabilité d'application de la règle $A \to B$.

Pour générer de nouveaux mots en utilisant cette grammaire, on part du symbole $\mathcal{S}$ et on applique successivement les règles jusqu'à obtenir un terminal. L'ensemble des mots qui peuvent être produits de la sorte s'appelle un langage, dans lequel à chaque mot est associée une probabilité égale au produit des probabilités des règles empruntées. L'algorithme de Weir a pour but de générer de nouveaux mots du plus probable au moins probable. Pour cela, l'algorithme maintient une file de priorité permettant d'avoir une bonne estimation du prochain mot du langage directement moins probable que le précédent. Une analyse plus approfondie de l'algorithme de Weir est effectuée dans le chapitre 6.

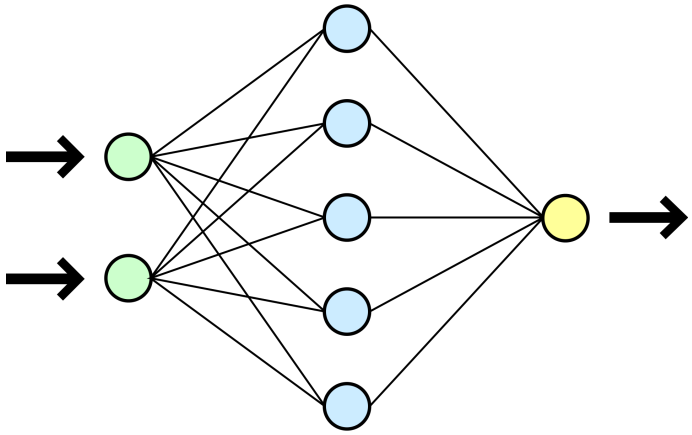


FIGURE 1.4 – Un exemple de réseau de neurones, avec $n = 3$ et $|\mathcal{N}| = 8$. Par Dake, Mysid. CC BY 1.0.
https://commons.wikimedia.org/w/index.php?curid=1412126

1.5.3 Apprentissage profond et réseaux de neurones

Les réseaux de neurones sont une technique d'apprentissage machine permettant d'approximer des fonctions de grande dimension. Le modèle consiste en un ensemble de neurones $\mathcal{N}$, répartis sur différentes couches $L_1, ..., L_n$, et où chaque neurone est connecté à tous ceux de la couche précédente et de la couche suivante. À chaque neurone est associée une fonction, dont les variables sont les images des fonctions des neurones de la couche précédente, et dont l'image sert pour la couche suivante. Chaque fonction possède également des paramètres pour pondérer les variables d'entrée afin de favoriser les plus intéressantes. L'apprentissage profond consiste à faire choisir de manière automatique les paramètres des fonctions en donnant des entrées au réseau et en lui indiquant quelle doit être la sortie.

Dans le cadre de la mesure de la robustesse des mots de passe, les réseaux de neurones sont utilisés pour prédire la suite d'un mot donné en entrée. De cette manière, si l'utilisateur commence à taper son mot et que le réseau de neurones prédit correctement les lettres suivantes, alors la robustesse du mot décroît. Pour cela, le réseau de neurones est entraîné sur un corpus de mots de passe. Ensuite, la robustesse d'un mot de passe est calculée en fonction du nombre de caractères correctement prédits par le réseau.

Les deux contributions majeures sur le sujet publiées au moment où cette thèse a débuté sont [35, 21], les résultats sont prometteurs mais encore à leurs balbutiements. Nos travaux sont compatibles avec les méthodes d'apprentissage profond, elles pourront donc être intégrées dans le futur si des résultats plus probants voient le jour.

Chapitre 2

Problématique et études réalisées

Dans ce chapitre, la problématique est tout d'abord exposée. Ensuite, les études réalisées durant la thèse sont présentées de manière plus détaillée.

Sommaire

2.1 Problématique

Il existe différentes manières de considérer la robustesse d'un mot de passe. On note s la fonction qui mesure la robustesse d'un mot m, $s(m)$ est donc la robustesse du mot m. Intuitivement, on peut penser qu'un humain est capable d'estimer $s(m)$ à l'œil nu. Pour une personne quelconque, le mot de passe "guizmo12" peut sembler faible, alors que le mot "K89Plkz427Fd" lui semblerait très fort. Cependant, rien n'indique que le deuxième mot n'est pas un mot partagé par beaucoup de comptes, auquel cas il suffirait de le trouver une seule fois pour casser tous les comptes qui l'utilisent. Ou bien il se peut que ce mot de passe soit aussi le nom de l'utilisateur. Dans ces cas, on peut penser que sa robustesse change. Cette mesure à l'œil nu considère que la robustesse est une propriété intrinsèque du mot de passe, et qu'il est donc possible de la mesurer uniquement en regardant ce mot de passe. Dans cette catégorie, on retrouve comme mesures de robustesse les entropies et les mesures heuristiques.

Une deuxième manière d'estimer $s(m)$ est de voir la robustesse d'un mot de passe comme une propriété contextuelle et non comme une propriété intrinsèque. On note donc $s(m|\mathcal{C})$ la robustesse de m selon le contexte $\mathcal{C}$. De ce point de vue,

on souhaite posséder le maximum d'informations contextuelles d'un mot de passe afin de mieux évaluer sa robustesse : le nom d'utilisateur associé, les informations personnelles de l'utilisateur, le service sur lequel il est utilisé... Lorsque l'attaquant utilise des informations contextuelles pour attaquer un mot de passe, il effectue une première étape, très importante, de collecte des données sur l'utilisateur, afin de faire grandir $\mathcal{C}$ le plus possible, tout en restant pertinent. Il construit ensuite un dictionnaire de candidats en fonction de ces données personnelles, afin d'essayer un à un les candidats. Même si la vision d'une robustesse intrinsèque semble la plus naturelle, les informations contextuelles permettent de trouver, en moyenne, le mot de passe beaucoup plus rapidement que sans information.

Une troisième manière d'estimer $s(m)$ est de considérer que m appartient à une base de données, $m \in \mathcal{D}$. On note $s(m|\mathcal{D})$ la robustesse du mot m sachant qu'il est dans $\mathcal{D}$. Ainsi, il est possible de définir un modèle probabiliste $\mathcal{M}$ associant des probabilités aux mots de $\mathcal{D}$. Plus un mot est probable d'après $\mathcal{M}$, moins il est robuste. Des énumérateurs peuvent être définis sur ce modèle probabiliste, permettant d'attaquer des mots de passe. C'est dans cette vision de la robustesse que les travaux de cette thèse sont réalisés.

La robustesse d'un mot de passe est également temporelle : un mot de passe fort aujourd'hui peut ne plus l'être plus tard. Par exemple, un mot de passe aléatoire de 8 caractères pouvait être considéré comme robuste dans les années 80 car la puissance de calcul des attaquants ne permettait pas en temps raisonnable de le trouver, mais aujourd'hui les puissances de calcul sont telles que ce mot de passe est obsolète. La mesure de la robustesse des mots de passe doit donc évoluer au cours du temps pour s'adapter aux nouvelles techniques et à l'augmentation de la puissance de calcul.

Les entropies traditionnelles de Shannon et Rényi ont été proposées comme mesures intrinsèques de la robustesse des mots de passe. Cependant, l'entropie permet de qualifier la quantité d'information délivrée par une source, alors qu'un mot de passe est le résultat d'un processus de génération, et non une source. L'entropie aurait donc du sens pour qualifier le mécanisme de génération du mot de passe, mais ne permet pas de mesurer la robustesse d'un mot de passe tout seul. Les entropies ont progressivement été abandonnées car elles sont très peu corrélées avec la difficulté à casser les mots de passe [4, 27, 53].

À la place il est devenu courant d'évaluer la robustesse des mots de passe en simulant l'attaque par un énumérateur, auparavant entraîné sur un jeu de données [37, 5, 54, 11]. Il y a deux avantages à cette manière de mesurer la robustesse. Premièrement, il est possible de calculer, d'après le modèle entraîné, la probabilité qu'un mot de passe soit choisi par un nouvel utilisateur. Deuxièmement, cela permet d'estimer la robustesse d'un mot de passe dans un scénario concret d'attaque. Cependant elle requiert de paramétrer correctement les énumérateurs, ce qui peut entraîner une mauvaise estimation de la robustesse s'ils sont moins bien entraînés que les attaquants réels.

Vient alors la question de l'évaluation de ces algorithmes. Puisqu'il existe plusieurs énumérateurs, basés sur différents modèles probabilistes, il devient nécessaire de les comparer pour comprendre les avantages et inconvénients de chacun. Actuellement, les énumérateurs sont comparés grâce à une métrique appelée "number of guesses"

qui définit le nombre de mots de passe trouvés dans un ensemble de mots ciblé en fonction du nombre de candidats générés par l'algorithme. Ainsi, pour chaque candidat généré, on compte le nombre de fois où il apparaît dans l'ensemble de mots ciblé. On estime que le meilleur énumérateur est celui qui a trouvé le plus de mots de passe en un certain nombre de candidats, représentant le coût que souhaite mettre l'adversaire dans son attaque.

2.2 Études réalisées

Afin de proposer une mesure de robustesse des mots de passe qui prenne en compte les nouvelles attaques probabilistes, il faut au préalable effectuer trois études : évaluer des performances des énumérateurs dans un contexte concret d'attaque, évaluer la pertinence des mesures de robustesses actuelles pour protéger des attaques probabilistes, et enfin comprendre de quelles manières il est possible de faire baisser les performance des énumérateurs probabilistes.

2.2.1 Performance des énumérateurs

La premier travail, situé au chapitre 4, consiste à évaluer ces attaques en simulant l'utilisation d'un logiciel de cassage de mots de passe, en mesurant la performance des énumérateurs dans ce scénario. Ce travail est nécessaire car il permet de mesurer la vitesse à laquelle un attaquant en pratique va casser des mots considérés, en fonction de paramètres qui auront été fixés au préalable. Parmi ces paramètres, on retrouve la fonction de hachage utilisée pour protéger les mots de passe de la base, le nombre de candidats que l'attaquant peut essayer dans une période, sa capacité à paralléliser les tâches, l'espace mémoire dont il dispose. Une fois ces paramètres fixés, il devient possible de lancer des expérimentations afin d'estimer la progression de l'attaquant en fonction du temps, et ainsi augmenter les paramètres de sécurité si l'on estime que sa progression est trop rapide. On peut imaginer qu'un administrateur système lance ce genre d'attaque sur la base de mots de passe qu'il administre afin de déterminer les éventuelles faiblesses tant au niveau de la robustesse des mots de passe que des paramètres de sécurité.

2.2.2 Efficacité des mesures heuristiques

La deuxième contribution, située au chapitre 5, consiste à évaluer l'efficacité des mesures heuristiques de robustesse contre les modèles probabilistes. L'hypothèse que l'on souhaite tester est que les politiques de composition des mots de passe actuellement déployées sur les services web ne sont pas toutes efficaces pour se protéger des énumérateurs probabilistes. Pour cela, on utilise les outils de la première contribution pour mesurer la performance des énumérateurs probabilistes dans des scénarios où les mots de passe sont filtrés par ces mesures heuristiques. Si l'hypothèse est vérifiée, cela confirme le fait qu'il est nécessaire de concevoir de nouvelles mesures de robustesse des mots de passe qui soient résistantes contre les énumérateurs

probabilistes. Même s'il paraît simple de prime abord de définir une telle mesure qui soit efficace contre un énumérateur donné, il est loin d'être évident qu'une même mesure soit efficace contre plusieurs énumérateurs.

2.2.3 Analyse du comportement des énumérateurs

Dans un troisième temps, il est nécessaire de comprendre par quels moyens il est possible de faire baisser la performance d'un énumérateur probabiliste. Ce travail est présenté dans le chapitre 6. Dans les deux précédentes contributions, les énumérateurs étaient étudiés en modèle boîte noire, c'est-à-dire que le fonctionnement interne n'était pas étudié. Cependant, il est essentiel d'observer ces énumérateurs en modèle boîte blanche en variant les bases de données d'entraînement afin de cerner la manière dont évoluent les structures de données utilisées. Puisque les énumérateurs probabilistes sont bien souvent des algorithmes qui parcourent des structures de données (listes, arbres, ...), la difficulté à générer des candidats se traduit par une augmentation de la taille de ces structures. Ainsi, en étudiant l'évolution de la taille de ces structures, il devient possible d'identifier les ensembles sur lesquels l'énumérateur est freiné. De ce fait, une mesure de robustesse doit faire en sorte d'augmenter le nombre de mots de passe dans ces ensembles pour qu'elle soit efficace contre ces énumérateurs probabilistes.

Chapitre 3

Travaux connexes

Dans ce chapitre, l'ensemble des travaux connexes à cette thèse sont présentés de manière à situer les contributions réalisées dans leur contexte.

Sommaire

3.1 Prémices

Dans cette section sont présentés les premiers travaux concernant la sécurité des mots de passe. À l'époque où les systèmes à temps partagés ont fait leur premiers pas, les besoins en sécurité pour les mots de passe commencent à émerger. La première publication à ce sujet date de 1979 [36] par Robert Morris et Ken Thompson, dans laquelle les auteurs présentent la manière dont sont stockés les mots de passe dans le système UNIX de l'époque : on sauvegarde le chiffré d'un message constant en utilisant comme clé le mot de passe avec l'algorithme DES. Même si les auteurs ne parlent pas de fonction de hachage, le principe est le même : ne pas avoir à stocker de clés afin qu'un attaquant ne puisse pas déchiffrer. Ainsi, pour retrouver le mot de passe associé à un compte en connaissant son chiffré, il existe deux solutions : effectuer une attaque à clair connu sur DES ou une force brute sur la clé de chiffrement. L'attaque à clair connu était probablement faisable à l'époque, mais peu de résultats probants avaient été publiés. Les humains ayant tendance à choisir des mots de passe courts et simples, la recherche de la clé de chiffrement étaient bien plus simple à effectuer.

Les auteurs alertaient déjà à l'époque sur le fait que le facteur critique d'une recherche de mot de passe est la quantité de temps nécessaire pour chiffrer un candidat et le vérifier. Ainsi, plus la fonction de chiffrement est rapide, plus l'attaquant peut essayer de mots en un temps donné.

De cet article se dégagent trois axes d'amélioration de la sécurité des mots de passe : utiliser une fonction de chiffrement lente, rendre les mots de passe moins prédictibles, et saler les mots de passe. Ces trois axes sont encore d'actualité même si davantage d'efforts sont mis sur la robustesse (non-prédictibilité) des mots de passe, sachant que des fonctions de hachage lentes et salant les mots de passe existent dans la littérature et sont satisfaisantes.

Dans la version 7 d'UNIX, publiée en 1979, la fonction *crypt(3)* est modifiée pour prendre en compte les conseils de cette publication. Le message constant est alors chiffré 25 fois avec les 8 premiers caractères du mots de passe comme clé (chaque caractère est codé sur 7 bits, ce qui donne une clé de taille 56 bits) en utilisant l'algorithme DES. Un sel de 12 bits est utilisé pour permuter la E-box de DES afin de rendre les pré-calculs difficiles.

Les fuites de bases de mots de passe dans la décennie 2010 montrent une dissonance entre l'avancement des trois axes dégagés par ce papier et l'usage des recommandations en pratique. En effet, il semblerait que les deux axes "ralentir le chiffrement" et "saler les mots de passe" sont encore aujourd'hui négligés malgré leur importance, puisque beaucoup de fuites de données montrent des mots de passe hachés avec des fonctions rapides et sans sel (voir figure 1.1). Pourtant, ces deux mesures sont simples à mettre en œuvre car des implémentations de fonctions lentes avec gestion du sel existent dans la plupart des langages et sont faciles à utiliser. En revanche, l'aspect "rendre les mots de passe moins prédictibles" est celui qui pose le plus de problèmes aujourd'hui tant en recherche qu'en pratique.

La deuxième publication importante au début des attaques sur les mots de passe est publiée en 1992 [28]. Ici, Klein se procure une liste de 13797 mots de passe hachés et salés avec la fonction de chiffrement présentée dans le précédent papier. L'objectif de l'auteur est de tester la résistance de ces mots de passe face à un attaquant qui souhaite en trouver le maximum. La stratégie utilisée dans ce travail est composée principalement d'attaques par dictionnaire avec des traitements supplémentaires tels que l'application de règles de réécriture et la combinaison de mots de différents dictionnaires. Après une attaque de presque 12 CPU mois (12 CPU pendant un mois ou 1 CPU pendant 12 mois), environ 25% des mots de passe ont été trouvés, dont 21% la première semaine. D'après ces statistiques, sur un système composé de 50 comptes, le premier mot de passe est cassé en moins de 2 minutes. Une fois que l'attaquant a mis un pied dans le système, même si le mot de passe root n'est pas cassé, il est libre d'exploiter des failles de sécurité pour élever ses privilèges. Dans ce papier, l'attaquant n'effectue pas de force brute, ce qui est dommage car il aurait été intéressant de voir à quel point les mesures conseillées par Morris et Thompson étaient efficaces.

Ce qui est intéressant dans cette contribution est le fait que déjà à l'époque, la puissance de l'attaquant était mesurée en temps de calcul (12 CPU mois), ce qui permettait de connaître concrètement de quoi l'attaquant était capable. Ceci

est important lorsque l'on doit choisir des paramètres de sécurité en fonction de la puissance de l'adversaire comme la difficulté dans les fonctions de hachage lentes. Comme le chantent un célèbre groupe de musique : "Know your enemy" [1].

À la suite de ces deux publications concernant les attaques sur les mots de passe, Massey publie un article en 1994 [34] où il cherche une relation entre l'entropie et le nombre d'essais nécessaires pour trouver un mot de passe ("guessing"). Pour cela il définit le modèle d'un attaquant souhaitant retrouver la valeur d'un mot de passe parmi un ensemble de possibilités, tel que défini dans la section 1.1.4.

Massey montre que la loi géométrique comme distribution de probabilité permet de maximiser l'entropie $H(X)$ où X est la variable aléatoire telle que définie dans la section 1.1.4. Il montre ensuite que $H(X)$ offre une borne exponentiellement inférieure à $E[G]$, telle que . Cela signifie, que moins les mots de $\mathcal{D}$ sont équiprobables, plus le nombre moyen d'essais nécessaires pour retrouver la valeur que prend X sera grand.

En revanche, Wang et al. [51] ont montré que la distribution des mots de passe générés par les utilisateurs suit plutôt une loi de Zipf, qui peut être paramétrée pour coller aux spécificités de la base de données en ne gardant que les mots de passe dont le nombre d'occurrences dépasse un seuil fixé, dépendant de la base.

3.2 Protection des mots de passe en base de données

La première manière historique de stocker les mots de passe en base de données (autrement qu'en clair) dans les systèmes UNIX était d'utiliser une fonction de chiffrement et de se servir du mot de passe comme clé. Cependant, la puissance de calcul des machines augmentait trop rapidement pour permettre aux utilisateurs d'adapter l'entropie et la longueur de leur mots de passe à une telle croissance. Il a donc été nécessaire de stocker les mots de passe de manière adaptative, c'est-à-dire en fonction de la puissance de calcul de l'attaquant. En 1999, Niels Provos et David Mazières proposent une nouvelle fonction de hachage dédiée au stockage de mots de passe appelée *bcrypt*. Elle utilise en interne la fonction de chiffrement Blowfish. *bcrypt* possède trois propriétés intéressantes pour stocker des mots de passe : elle est lente, elle intègre la gestion du sel, et elle est adaptative.

Elle est lente. L'algorithme de *bcrypt* commence avec une étape d'initialisation de l'état des clés de l'algorithme Blowfish, appelée *EksBlowfishSetup*. C'est cette étape qui est coûteuse car elle effectue un grand nombre d'expansions de clé.

Elle intègre la gestion du sel. Le sel est utilisé lors des expansions des clés, et doit être passé en paramètre de la fonction de hachage. De ce fait, il n'est pas nécessaire au développeur de gérer lui même le sel, évitant ainsi des erreurs qui ont conduit à des faiblesses de sécurité dans le passé.

Elle est adaptative. Le nombre d'expansions de clés successives dans l'étape *EksBlowfishSetup* est déterminé par un paramètre de *bcrypt* appelé le coût. L'initialisation des clés est répétée $2^{coût}$ fois. Ainsi, il est possible, de manière simple, d'augmenter le coût nécessaire pour hacher un mot de passe en fonction des ressources

1. Pas Green Day.

de l'attaquant au cours du temps. Cela nécessite cependant de connaître le mot de passe afin de calculer la nouvelle empreinte.

bcrypt est intéressante à utiliser pour stocker les mots de passe, car elle permet de ralentir grandement la vitesse des attaques : par exemple, sur une carte graphique GTX 1080 Ti [19], l'attaquant est capable de hacher 2×10^4 mots par seconde avec *bcrypt*(coût 5, 600 avec un coût de 10), alors qu'il en hache 10^9 avec SHA-1. Aussi, le fait qu'*EksBlowfishSetup* modifie constamment les S-boxes utilisées empêche leur partage par plusieurs instances. Il est donc nécessaire d'avoir autant de copies des S-boxes, de taille 4KB, en mémoire que d'instance de mot à hacher en simultané. Même si aujourd'hui il reste envisageable sur un ordinateur de prévoir 4GB de mémoire pour hacher 10^6 mots en même temps, cela devient coûteux de créer des circuits imprimés (ASIC, FPGA) avec autant de mémoire. Même s'il existe des ASIC pour miner les cryptomonnaies utilisant scrypt, le rapport performances/prix des CPU et GPU pour miner est meilleur. Ce genre de circuit est très efficace contre des mots de passe hachés avec des fonctions rapides et peu coûteuses en mémoire comme SHA-1 ou SHA256.

Au début des année 2000, les premiers FPGA et ASIC font leur apparition pour casser des mots de passe [12]. Ces matériels sont des circuits intégrés programmés pour calculer un nombre restreint de fonctions à très grande vitesse. Le problème de *bcrypt* est que le coût en mémoire pour hacher un mot de passe est constant, il ne peut donc pas être adapté en fonction des ressources de l'adversaire. Ceci implique qu'au lieu de rendre plus rapide les FPGA/ASIC pour exécuter *bcrypt*, il est plus intéressant d'agrandir les circuits pour exécuter plusieurs *bcrypt* en parallèle, sans devoir augmenter la mémoire de manière conséquente. Par exemple, pour cracker les mots de passe deux fois plus vite, il suffit de dupliquer le circuit sur la même carte.

Une manière naturelle de freiner cet avantage de l'attaquant est d'augmenter la taille des circuits. Ainsi il deviendra plus coûteux de paralléliser les exécutions des fonctions de hachage. Pour cela, Colin Percival introduit la notion de fonction "memory-hard" ainsi que *scrypt* [41], une fonction de hachage dont l'espace mémoire et le temps nécessaires à une exécution sont paramétrables.

Plus récemment, Argon2 [2] a été développée, gagnant de la Password Hashing Competition [1] en 2015. C'est une fonction de hachage "memory-hard" dont le temps, l'espace et le nombre de cœurs requis pour une exécution sont paramétrables. Elle possède deux variantes : Argon2d et Argon2i. Argon2d ("d" pour dépendant) est plus rapide, utilise des zones mémoire qui sont dépendantes des données d'entrée et est donc vulnérable aux attaques par canaux cachés. Elle est adaptée pour les cryptomonnaies et les applications qui ne sont pas susceptibles d'être attaquées par canaux cachés. Argon2i (pour indépendant) utilise des zones mémoires de manière indépendante des données d'entrée, ce qui la protège des attaques par canaux cachés. Elle est plus lente, et les auteurs recommandent cette variante pour hacher les mots de passe.

Il est recommandé aujourd'hui d'utiliser *Argon2* car son paramétrage peut-être effectué plus finement. Il est possible d'ajuster la mémoire requise de manière indépendante du temps, ce qui n'est pas le cas de *scrypt*.

3.3 Métriques de robustesse

Dans cette section sont présentés les principaux travaux proposant des métriques de robustesse des mots de passe, ainsi que les principaux travaux qui comparent ces métriques ou étudient leur efficacité pour protéger des mots de passe contre des attaques contemporaines.

3.3.1 Métriques heuristiques

Une métrique est heuristique si elle ne considère que le contenu du mot afin d'en évaluer la robustesse. Cela vient en opposition aux métrique statistiques qui observent la distribution des mots de passe pour mesurer la robustesse.

3.3.1.1 ZXCVBN

Une métrique heuristique plus sophistiquée que celles que l'on trouve généralement dans les services en ligne est celle proposée par Wheeler [55] pour le service Dropbox, appelée *ZXCVBN* (une suite de touches adjacentes d'un clavier en disposition QWERTY). Cette métrique essaie de reconstituer la manière dont le mot de passe a été construit, et prend en compte plusieurs attaques différentes. Pour cela, elle découpe le mot en jetons (tokens) selon une liste de stratégies : mots dans une liste, dates, nombre, leetspeak, ... Ensuite, pour chaque jeton, elle estime le nombre d'essais à chaque attaque pour le deviner. Pour une attaque par dictionnaire, le nombre d'essais est le rang du jeton dans la liste, pour une attaque par force brute, le nombre d'essais est la moitié (en moyenne) du cardinal de l'ensemble des mots de même longueur. Le nombre d'essais du mot est estimé comme étant la somme du nombre d'essais pour chaque jeton. La métrique attribue un score entier entre 0 et 4 en fonction du nombre d'essais estimé. Les scores 0, 1, 2 et 3 correspondent à un nombre d'essais estimé à moins de $10^3, 10^6, 10^8$ et 10^{10} respectivement, et le score 4 au delà. ZXCVBN se veut léger et utilisable dans un navigateur web, ce qui en fait sa popularité.

3.3.2 Métriques statistiques

Les travaux proposant des métriques de robustesse statistiques des mots de passe sont présentés dans cette partie. Les métriques statistiques sont des métriques qui évaluent la robustesse des mots en fonction de données observées (fuites de mots de passe, sondages, ...). Elles viennent en opposition aux métriques heuristiques qui ne prennent en compte que la composition du mot.

Castelluccia et al. [7] proposent une métrique adaptative de robustesse des mots de passe fondée sur un modèle de chaîne de Markov. Une mesure est dite adaptative si elle évolue au cours du temps pour un mot de passe donné. La mesure proposée requiert de maintenir une table où l'on compte le nombre d'occurrences de chaque n-gram sur l'ensemble des mots de passe. Ainsi, lorsqu'un nouvel utilisateur s'enregistre, il est possible de calculer un score de robustesse en fonction de la distribution des n-grams observés auparavant. Dans le cas où le score de robustesse calculé est inférieur à un

seuil fixé, le mot de passe est rejeté. Sinon, il est accepté et la table est mise à jour. Un bruit est ajouté lors de la mise à jour de la table afin de garantir qu'une éventuelle fuite de cette table apporte le minimum d'information possible à l'attaquant. Cette méthode est intéressante, car elle permet de contrôler plus finement la popularité des n-grams, et ainsi éviter des mots de passe construits à partir d'un même mot. Cependant, elle n'est pas suffisante car elle ne permet pas d'estimer la capacité de l'attaquant à casser les mots de passe, et ne permet pas non plus de considérer un attaquant utilisant un autre modèle probabiliste, comme une grammaire.

Une métrique similaire à celle proposée par Castelluccia est PathWell, proposée par KoreLogic, une entreprise de sécurité informatique américaine. PathWell fonctionne sur le même principe que la métrique de Castelluccia, mais avec les bases composées d'une grammaire de Weir. Pour cela, on compte le nombre d'occurrences de chaque base composée. Ainsi, à chaque fois qu'un mot de passe est entré dans le système, sa base composée est calculée (par exemple $L_6D_2S_1$) et son nombre d'occurrences est incrémenté. On peut ensuite paramétrer le système pour interdire les bases composées trop populaires, obligeant les utilisateurs à en utiliser une moins populaire. De la même manière que la métrique de Castelluccia, PathWell est intéressante mais ne suffit pas pour se protéger d'un attaquant qui utiliserait les chaînes de Markov pour attaquer les mots de passe.

Une métrique qui correspond aux attentes de cette thèse est celle de Kelley et al. [27]. Dans ce travail, les auteurs mettent en place une métrique de robustesse qui calcule un score en simulant deux attaques : une basée sur les modèles de Markov, et l'autre utilise les grammaires probabilistes de Weir. En ayant une estimation du nombre d'essais nécessaires de ces deux attaques pour casser le mot de passe en entrée, le score est calculé en un temps raisonnable car les attaques sont simulées. De plus, les attaques peuvent être paramétrées en entraînant les modèles probabilistes associés sur les données souhaitées, permettant ainsi de s'adapter au public du service l'utilisant. Simuler une attaque supplémentaire demande du travail mais permet de s'intégrer facilement à la solution proposée. Le principal inconvénient d'une telle solution est l'espace nécessaire pour stocker les modèles entraînés, ce qui rend difficile son utilisation dans un navigateur web.

Dell'Amico et Filippone [10] améliorent le principe en estimant le rang d'énumération d'un mot de passe via une méthode de Monte Carlo : un échantillonnage des candidats est effectué afin d'obtenir une estimation d'où se situe le mot de passe en fonction des candidats tirés aléatoirement. De cette manière, il est possible d'estimer le rang même de mots requérant 2^{80} essais pour être trouvés, ce qui n'est pas le cas de la métrique précédente, qui elle est plus coûteuse en temps et en espace. Dans les travaux de cette thèse, il n'a pas été nécessaire d'estimer le rang des mots aussi loin que 2^{80}, c'est pourquoi cette méthode n'a pas été utilisée.

3.3.3 Métrique multimodale

Une métrique multimodale est un regroupement de plusieurs métriques spécifiques, et dont le but est de couvrir les défauts de chacune des métriques avec les qualités des autres, par exemple associer des métriques heuristiques et des métriques statistiques.

C'est probablement l'approche la plus intéressante en terme d'efficacité. Cependant, concevoir de telles métriques demande un travail plus approfondi pour plusieurs raisons :

— il faut s'assurer que les métriques sont compatibles entre elles, c'est-à-dire que leur regroupement nuise un minimum l'utilisabilité du système ;

— évaluer leur efficacité nécessite de montrer que c'est bien le regroupement de ces métriques qui font que le système fonctionne, et non uniquement un sous-ensemble de ces métriques ;

— il faut que l'implantation d'une métrique multimodale reste simple, car complexité rime souvent avec biais et faille de sécurité. De plus, si l'on veut que les services l'adoptent, il faut soit fournir une implantation facile à comprendre, soit permettre aux développeurs de l'écrire facilement (par exemple, proposer une Request For Comments).

Dans leur publication en deux parties, Galbally, Coisel et Sanchez [15, 16] définissent ce qu'est une métrique multimodale et quelles doivent être ses propriétés. Ils découpent en deux temps l'évaluation de la robustesse d'un mot de passe :

1. évaluation d'un mot trivial : un mot est trivial s'il est trouvable en un nombre limité d'essais ;
2. évaluation d'un mot non-trivial : un mot n'est pas trivial s'il doit être trouvé par une attaque plus sophistiquée basée sur un modèle probabiliste.

Ils proposent enfin un métrique de robustesse multimodale en 4 modules :

1. résistance à une attaque par dictionnaire ;
2. résistance à une attaque par force brute ;
3. résistance à une attaque basée sur une chaîne de Markov à mémoire adaptative ;
4. résistance à une attaque basée sur une chaîne de Markov hiérarchique.

Les deux premiers modules sont simples à implanter, et ne nécessitent que de choisir un bon dictionnaire. Cependant, les deux derniers modules sont probablement plus complexes à implanter.

L'attaque du module 3 est dite à mémoire adaptative car la mémoire de la chaîne de Markov n'est pas fixée (le paramètre n). L'attaque du module 4 est dite hiérarchique car elle procède en plusieurs étapes : elle estime d'abord la probabilité d'occurrence des structures de plus haut ordre (à la manière d'une grammaire probabiliste) pour ensuite estimer la probabilité de chaque structure. Par exemple, l'enchaînement des différentes classes de caractères dans un mot de passe peut également être représenté par une chaîne de Markov, et chaque suite de caractères d'une même classe par une autre chaîne de Markov.

3.3.4 Études de mesures existantes

Dans cette sous-section sont présentés les travaux qui testent l'efficacité des mesures de robustesse face à des attaques de l'époque.

Komanduri et al. [30] étudient les effets des politiques de création de mots de passe sur le comportement des utilisateurs et la robustesse de leurs mots de passe. En étudiant le comportement des utilisateurs, ils remarquent que la politique de création qui demande un mot de passe de taille 16 minimum permet d'avoir à la fois un bon taux d'acceptation du premier mot de passe, peu de réutilisation et assez peu de frustration de la part des utilisateurs. C'est pour cette raison que cette politique a été choisie pour les travaux de la deuxième contribution.

Dans une publication de 2015, Ur et al. [48] comparent l'efficacité des énumérateurs probabilistes pour casser des mots de passe selon différents contextes. Ils évaluent également l'efficacité de l'attaque réalisée par des professionnels de la récupération de mots de passe. Ils montrent que les professionnels sont très souvent meilleurs que les énumérateurs probabilistes utilisés avec les paramètres par défaut, mais que s'ils sont bien paramétrés et utilisés ensemble, ces énumérateurs peuvent être quasiment aussi efficaces que les professionnels. Les résultats montrent également que les énumérateurs ne peuvent pas être efficaces sur toutes les données. En effet, pour chaque énumérateur, il existe des ensembles de mots qu'il a du mal à casser. Ces résultats affirment le fait qu'il ne faille pas se reposer sur un unique énumérateur pour évaluer la robustesse des mots de passe.

Segreti et al. [45] évaluent l'efficacité des politiques de composition adaptatives. Pour cela, ils conduisent une étude en utilisant la métrique proposée par KoreLogic [31] et montrent que ces métriques adaptatives apportent une sécurité conséquente sans pour autant incomber sur l'utilisabilité. La métrique PathWell de KoreLogic conserve la liste des bases composées (au sens de la grammaire de Weir, c'est-à-dire la suite des classes de caractères) utilisées pour les mots de passe. Si une structure devient trop utilisée, elle est bannie, obligeant les nouveaux utilisateurs à en utiliser une autre.

Golla et al. [17] proposent une manière d'évaluer la précision des métriques de robustesse des mots de passe en calculant des similitudes entre une base de référence et les scores retournés par les métriques sur une base de test. Ils montrent que les métriques académiques basées sur des modèles probabilistes tels que les chaînes de Markov, les grammaires probabilistes ou les réseaux de neurones récurrents sont les plus précises et efficaces pour mesurer la robustesse des mots de passe. Ceci confirme le fait qu'il faille s'en servir pour se protéger des attaquants utilisant des énumérateurs probabilistes pour attaquer les mots de passe.

3.4 Modèles probabilistes

La première publication de l'utilisation d'un modèle probabiliste pour casser des mots de passe est en 2005, où les auteurs présentent un compromis temps-espace qui utilise les chaînes de Markov [37]. Les modes Markov de John The Ripper et Hashcat sont basés sur ce modèle.

Les algorithmes d'OMEN et celui de la grammaire de Weir sont étudiés plus en détails dans le chapitre 6, où leur comportement est analysé dans le but de mieux protéger les bases de données contre des attaques les utilisant.

3.4.1 OMEN

En 2015, Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia et Daniele Perito proposent OMEN [11], Ordered Markov ENumerator, un algorithme basé sur les chaînes de Markov. Cet énumérateur est une amélioration du compromis temps-espace de 2005 qui le rend plus performant, en approximant les probabilités des candidats afin de gagner en espace et en temps. Pour cela, OMEN va ranger chaque n-gram dans une case en fonction de sa probabilité. Les cases sont numérotées de 0 à -9 (les nombres sont négatifs), et les n-gram les plus probables sont rangés dans des cases proches de 0. La répartition des n-gram dans les 10 cases n'est pas uniforme mais logarithmique, c'est-à-dire que la cardinal des cases augmente quand on s'approche de -9.

Pour générer des candidats de taille l fixe, OMEN choisit $l - n + 1$ cases. Il va ensuite générer tous les chevauchements possibles des n-grams des cases qu'il a choisies. Au début de l'énumération, OMEN choisira des cases proches de 0, car elles permettent de générer des candidats plus probables que les autres. Puis au fur et à mesure de l'énumération, OMEN choisira des cases dont le numéro est plus petit. Les candidats seront alors moins probables mais en plus grand nombre.

Le fait que plusieurs n-gram de probabilités différentes soient rangés dans la même case (procédé appelé "smoothing") ne permet plus d'ordonner parfaitement les candidats, cependant, l'ordre dans lequel ils sont générés reste acceptable par rapport au gain de performance et d'espace que ce compromis apporte.

De plus, OMEN s'adapte au succès des mots générés. En effet, il ordonne les ensembles de mots à générer en fonction du taux de succès des mots de même longueur. Par exemple, si les mots de taille 8 qu'il a généré ont permis d'avoir un meilleur taux de succès que ceux d'une autre taille, il continuera à générer des mots de taille 8. L'avantage d'avoir une stratégie adaptative est de ne pas passer trop de temps à énumérer des candidats qui ne permettent pas de trouver beaucoup de mots de passe. Un désavantage est que cela rend difficile de calculer le rang d'un mot dans l'énumération, puisque cela dépend de la base attaquée.

3.4.2 Grammaire de Weir

En 2009, Matt Weir propose d'utiliser les grammaires probabilistes pour attaquer les mots de passe [54]. Pour cela, il utilise les grammaires probabilistes non récursives et non ambiguës, telles que définies dans la partie 1.5.2.2. Les terminaux produits par ces grammaires sont ainsi utilisés comme candidats dans le processus de cassage des mots de passe.

Pour produire les terminaux de ces grammaires de manière efficace, l'auteur utilise une structure de données, la file de priorité, pour parcourir la grammaire.

Afin de mesurer l'efficacité de l'utilisation de cet algorithme dans le cadre des attaques sur les mots de passe, la grammaire est entraînée sur une partie d'une base de données de mots de passe ayant déjà fuitée. Ensuite, l'algorithme de Weir parcourt cette grammaire et on mesure le nombre de mots de passe trouvés dans l'autre partie de la base de données, en fonction du nombre de mots générés par l'algorithme. On

trace ensuite la courbe du nombre de mots trouvés en fonction du nombre de mots générés. L'efficacité de l'algorithme de Weir est comparée à celle du mode Markov de John The Ripper.

Les résultats montrent que l'algorithme de Weir est bien meilleur que le mode Markov de John The Ripper car il trouve environ 50% de mots de passe en plus en 3×10^8 candidats. De plus, l'algorithme de Weir est d'autant plus efficace que les mots sont longs car la taille des candidats est prise en compte, alors que JtR les génère par taille croissante ce qui devient très coûteux vers la fin.

Le principal inconvénient à l'utilisation de cet algorithme est l'espace mémoire utilisé par la grammaire et par la file de priorité. En effet, plus l'attaque sera longue et plus la file de priorité sera grande. Il peut donc être intéressant d'étudier des compromis mémoire/précision afin de réduire la taille de la file tout en garantissant une efficacité raisonnable.

3.4.3 Grammaire de Komanduri

Dans son manuscrit de thèse, Saranga Komanduri [29] propose une amélioration des grammaires de Weir. Premièrement, il suggère d'apprendre les fréquences des règles de la forme $L_i \rightarrow w$ où w est un terminal, depuis les données d'apprentissage car à l'origine, Weir attribuait une probabilité uniforme à ces règles pour des raisons de simplicité. Cette amélioration permet d'être plus fidèle à la réalité mais aussi de réduire le nombre de terminaux et donc l'espace mémoire requis. Dans un deuxième temps, pour encore réduire l'espace mémoire utilisé, l'auteur fait une approximation des probabilités des terminaux ce qui permet de les regrouper plus facilement. La troisième amélioration est la possibilité de produire des terminaux qui ne sont pas présents dans les données d'apprentissage, en leur assignant une probabilité uniforme. Enfin, il propose de complexifier la grammaire utilisée en ne produisant plus uniquement que des terminaux ne comportant qu'une seule classe de caractères mais en autorisant des terminaux composés de plusieurs classes de caractères.

3.5 Énumérateurs non-probabilistes

3.5.1 PRINCE

PRINCE [46] est un énumérateur non-probabiliste qui prend en entrée une liste de mots, et calcule les produits cartésiens des partitions de mots de même longueur. Dans un premier temps, chaque mot $w = c_1...c_k$ de l'ensemble d'entrée $\mathcal{W}$ est rangé dans la partition P_k regroupant les mots de taille k. Ensuite, étant donnée la longueur maximum l des candidats que l'on souhaite générer, PRINCE calcule toutes les partitions d'entiers possibles $a_1, a_2, ..., a_n$ telles que $\sum_{i=1}^{n} a_i \leq l$. Chaque possibilité $s = a_1, ..., a_n$ est appelée une chaîne. PRINCE calcule le cardinal de chaque chaîne tel que

$$S = Card(s = a_1, ..., a_n) = \prod_{i=1}^{n} Card(P_{a_i}).$$

Les s sont ensuite ordonnées par ordre croissant de cardinalité S. Pour chaque $s = a_1, ..., a_n$ dans l'ordre, PRINCE calcule le produit cartésien Φ_s des ensembles $P_{a_1}, ..., P_{a_n}$, et produit les mots issus de la concaténation des mots de chaque n-uplet de Φ_s, sans permutation. Par exemple, si on considère la chaîne $s = 2 + 3 + 4$, alors PRINCE générera toutes les concaténations possibles des mots de taille 2, 3 et 4, sans permutation.

Chapitre 4

Mesure de la performances des énumérateurs

L'objectif de ce chapitre est de formaliser et mesurer la performance des énumérateurs de mots de passe. Pour cela, il a été nécessaire de modéliser le processus d'attaque hors-ligne sur les mots de passe protégés par une fonction de hachage. Une formalisation du calcul de la performance de ce processus est ensuite exposée. La mesure des différentes quantités afin de calculer cette performance a demandé de résoudre des challenges techniques qui n'avaient pas été présentés dans la littérature, nécessitant de définir la performance en fonction du taux de succès et de la vitesse du processus. Les expériences menées montrent très clairement que l'énumérateur PCFG est meilleur dans presque toutes les conditions testées, même quand la fonction de hachage est rapide, par exemple SHA-1. Les énumérateurs rapides et optimisés pour la cassage de mots de passe tels que le mode Markov de John The Ripper et la force brute ont tout de même de bonnes performances lorsque la fonction de hachage attaquée est rapide. Au contraire, les énumérateurs probabilistes deviennent bien plus performants lorsque la fonction de hachage est lente, telle que bcrypt. Il est donc nécessaire, si les énumérateurs ont pour objectif de casser des mots de passe, qu'ils soient évalués en tenant compte du processus complet et en particulier de la vitesse à laquelle ils génèrent des candidats.

Sommaire

4.1 Introduction

Ce chapitre présente trois contributions. Tout d'abord, la modélisation du processus de cassage des mots de passe ainsi que la formalisation de sa performance sont présentées, en tenant compte du coût de traitement des candidats. Deuxièmement, puisque la mesure de la performance ne peut pas être directement effectuée, il est nécessaire de conduire deux mesures distinctes du taux de succès et de la fréquence du processus. Il se présente un effet similaire à celui d'Heisenberg : mesurer la fréquence a un impact sur sa valeur, ce qui demande de mettre en œuvre une technique appropriée pour effectuer cette mesure. La troisième contribution concerne les résultats des expériences sur les bases de données fuitées. Les résultats confirment que les fonctions de hachage cryptographiques rapides sont inadaptées pour le stockage des mots de passe, puisque même les stratégies les plus naïves telle que la force brute sont très performantes en les attaquant. Ces expériences montrent également que les énumérateurs académiques sont très utiles pour attaquer des bases de données protégées avec des fonctions de hachage lentes. Ces fonctions sont donc nécessaires mais pas suffisantes pour protéger les mots de passe de ces attaques. Pour rendre ces énumérateurs probabilistes encore plus efficaces, les développeurs devraient les implanter dans les logiciels de cassage de mots de passe.

Les contributions de ce travail permettent de bien choisir une fonction de hachage en connaissance des menaces en termes d'énumérateurs (côté défense) mais aussi pour choisir un bon ensemble d'énumérateurs par rapport à la fonction de hachage utilisée (côté attaque, casseurs de mots de passe, pentesteurs).

Pour mesurer l'impact des fonctions de hachage, nous souhaitons mesurer les performances du système complet de cassage des mots de passe, comprenant la génération des candidats, leur hachage et la recherche de leur empreinte dans la base cible. La performance sur une période de temps donnée est définie comme le nombre de mots de passe trouvés dans cette période.

Comme présentés dans le chapitre 1.5, les énumérateurs probabilistes académiques tentent d'ordonner les candidats générés par probabilité décroissante au détriment de la vitesse à laquelle ils les génèrent. D'un autre côté, les énumérateurs plus rustiques implémentés dans les logiciels de cassage tels que John The Ripper et Hashcat, privilégient la vitesse de génération des candidats sans le faire dans un ordre décroissant de probabilité. Aussi, la fonction de hachage est déterminante pour la vitesse du système complet, car plus elle est rapide plus on peut tester de candidats dans une période donnée. La qualité des candidats générés par les énumérateurs probabilistes est meilleure, mais ils sont plus lents à produire. Il faut donc trouver un compromis entre la vitesse d'énumération des candidats et leur qualité. Nous savons que la fonction de hachage a une influence sur la performance du système complet, cependant nous souhaitons mesurer cet impact pour avoir une vision plus fine du compromis vitesse/qualité. Par exemple, jusqu'à quelle lenteur de fonction de hachage la force brute est-elle une méthode viable ?

Ce chapitre est organisé de la façon suivante : premièrement, le contexte et les choix de fonction de hachage et des bases de données sont rapidement exposés. Ensuite, la modélisation du processus de cassage des mots de passe est présentée en

section 4.3. En section 4.4 est présentée la manière dont la performance est calculée en fonction des mesures du taux de succès et de la fréquence. En section 4.5 sont présentés les résultats des mesures de performance sur deux base de données en simulant deux fonctions de hachage différentes. La conclusion est ensuite développée.

4.2 Contexte

Dans ce travail, nous considérons deux fonctions de hachage : la première, SHA-1, de part sont usage massif dans les fuites de données récentes. SHA-1 sera une représentante de la classe des fonctions cryptographiques rapides (MD5, SHA-1, SHA2, SHA-3, ...), puisque dans cette étude ces fonctions ont un comportement similaire. La deuxième fonction utilisée est bcrypt, une référence pour stocker les mots de passe, avec un coût de 10 ce qui est la valeur par défaut de la fonction *password_hash()* de PHP au moment de l'écriture. Argon2, Scrypt et bcrypt sont également très similaires dans le cadre de ce travail, c'est pourquoi bcrypt sera la représentante de sa classe car elle est davantage utilisée que Argon2 et Scrypt, mais aussi parce que le seul paramètre qui nous intéresse est le coût en temps d'une exécution. Dans les expériences qui suivent, deux des bases de données présentées en section 1.1.6 sont utilisées : **LinkedIn** et **Rockyou**.

Lors de nos expériences, nous avons rapidement remarqué que la fréquence de génération des candidats par les énumérateurs n'était pas constante. Même si l'on a l'impression de prime abord que cette fréquence varie peu au début de l'énumération, on remarque néanmoins des périodes où la fréquence va lourdement chuter, provoquant des pics de baisse de fréquence. Ceci s'explique par le fait que les énumérateurs vont souvent devoir changer l'ensemble des mots sur lesquels ils travaillent, impliquant un nombre important d'opérations à effectuer pour passer d'un ensemble à un autre. Aussi, plus l'on avance dans l'énumération et plus les structures de données utilisées grandissent, demandant davantage d'opérations pour les parcourir. Il est donc faux d'estimer que cette fréquence est constante sur l'ensemble de l'énumération.

4.3 Modélisation

4.3.1 Contexte

Le processus d'attaque sur les mots de passe, au niveau du traitement d'un seul candidat, se résume en 3 étapes (voir Figure 4.1) :

(i) générer un candidat c

(ii) le hacher en calculant $H(c)$

(iii) chercher une correspondance de l'empreinte $H(c)$ dans la base cible D

En pratique, l'étape (iii) est négligeable, même lorsqu'une fonction de hachage rapide est utilisée. En effet, les logiciels de cassage de mots de passe utilisent une structure probabiliste, similaire à un filtre de Bloom avec une seule fonction de hachage, pour stocker les empreintes attaquées en mémoire. Nous négligerons donc

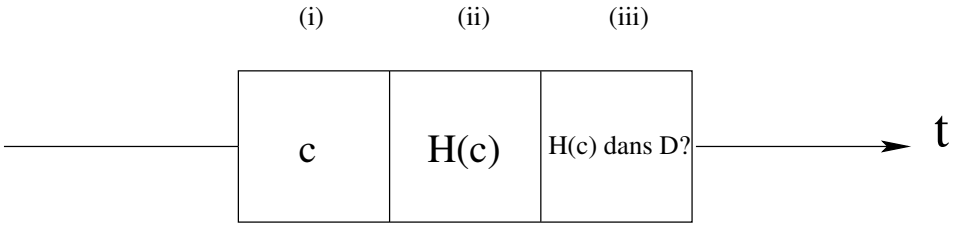


FIGURE 4.1 – Processus de traitement d'un candidat

FIGURE 4.2 – Processus de génération de plusieurs candidats en fonction du temps

cette étape. Ainsi, selon la fonction de hachage utilisée dans l'étape (*ii*), le goulot d'étranglement du système est soit l'étape (*i*), soit l'étape (*ii*).

À une échelle plus large et donc au traitement de plusieurs candidats, le processus peut être représenté comme sur la Figure 4.2. Les c représentent la génération d'un candidat et les p signifient que le candidat correspondant est un mot de passe présent dans la base de données attaquée.

4.3.2 Formalisation de la performance

Le processus de cassage des mots de passe est un processus continu. Il y a donc un besoin de formaliser la notion de performance pour une attaque sur les mots de passe.

Considérons dans un premier temps la performance au niveau de la génération d'un seul candidat. Pour tout $i \in \mathbb{N}$, notons t^i l'instant où le traitement du $i^{éme}$ candidat est terminé, g^i le gain de ce candidat, *i.e.* le nombre de fois qu'il apparaît dans la base de données D, et c^i le temps requis pour traiter ce candidat (les 3 étapes), *i.e.* $c^i = t^i - t^{i-1}$. La performance est donc $P(t^{i-1}, t^i) = g^i$. Ainsi

$$P(t^{i-1}, t^i) = c^i \, \frac{g^i}{c^i} = \int_{t^{i-1}}^{t^i} \frac{g(t)}{c(t)} dt,$$

où $c(t)$ et $g(t)$ sont constants sur $]t^{i-1}, t^i]$.

Ce modèle peut également être utilisé s'il y a une parallélisation, puisque $g(t)$ et $c(t)$ peuvent être mesurés sur plusieurs cœurs.

Soit t_1 et t_2 deux instant tels que $t_1 < t_2$. La performance dans cette période $]t_1, t_2]$ est donc

$$P(t_1, t_2) = \int_{t_1}^{t_2} \frac{g(t)}{c(t)} dt,$$

où $c(t) = c^i$ et $g(t) = g^i$, pour tout $t^{i-1} < t \leq t^i$.

La fréquence $F(t_1, t_2)$ est par définition le nombre de candidats traités entre t_1 et t_2. On a

$$F(t_1, t_2) = \int_{t_1}^{t_2} \frac{1}{c(t)} dt.$$

Le taux de succès $S(t_1, t_2)$ est défini comme le ratio entre le nombre de mots trouvés dans la période $]t_1, t_2]$ et le nombre de candidats traités dans le même temps :

$$S(t_1, t_2) = \frac{P(t_1, t_2)}{F(t_1, t_2)}.$$

Comparaison des performances d'énumérateurs. Il existe deux cas qui dépendent de la fonction de hachage choisie. Pour chaque candidat i, on a $c^i = c_g^i + c_h^i + c_d^i$ où c_g^i correspond à l'étape (i), c_h^i à l'étape (ii), et c_d^i à l'étape (iii), avec c_d^i négligeable en pratique. Soient E_1 et E_2 deux énumérateurs dont on souhaite comparer les performances. On a deux cas :

— **cas a) une fonction de hachage lente est utilisée.** Alors pour tout i on a

$$c^i = c_g^i + c_h^i \approx c_h^i$$

et

$$F_1(t_1, t_2) \approx F_2(t_1, t_2),$$

et si

$$S_1(t_1, t_2) > S_2(t_1, t_2)$$

alors on a

$$P_1(t_1, t_2) > P_2(t_1, t_2),$$

c'est-à-dire que l'énumérateur E_1 a une meilleure performance que E_2 quand E_1 a un meilleur taux de succès.

— **cas b) une fonction rapide est utilisée.** Alors pour tout i on a

$$c^i = c_g^i + c_h^i \approx c_g^i,$$

et si

$$\frac{S_1(t_1, t_2)}{c_{g,1}} > \frac{S_2(t_1, t_2)}{c_{g,2}}$$

alors on a

$$P_1(t_1, t_2) > P_2(t_1, t_2).$$

C'est à dire que l'énumérateur avec le meilleur ratio taux de succès sur temps de génération des candidats est le plus performant. Il y a donc ici un compromis entre le taux de succès et la vitesse de l'énumérateur.

Les travaux de la littérature calculant le taux de succès à partir du début de l'énumération sans tenir compte de la fréquence correspondent au cas a), quand une fonction lente est utilisée. En effet dans ce cas, la vitesse d'énumération n'est pas vraiment importante et les meilleures performances sont obtenues grâce à un meilleur taux de succès. Dans le cas b), lorsqu'une fonction rapide est utilisée, l'énumérateur

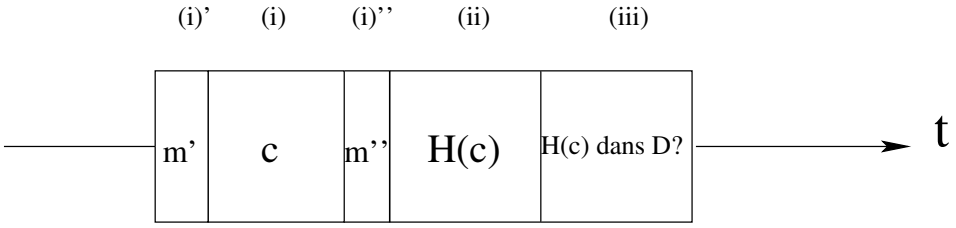


FIGURE 4.3 – Processus de génération d'un candidat avec mesure du temps d'exécution

E_1 ne doit pas uniquement avoir un meilleur taux de succès que E_2 mais aussi être plus rapide, ou alors que son taux de succès contre-balance la différence de vitesse entre les deux. Par exemple, si E_2 a un taux de succès deux fois plus grand que E_1 mais est cent fois plus lent, les performances de E_2 seront cinquante fois plus petites pour la même période de temps. Dans ce cas, mesurer uniquement le taux de succès pour comparer leur performance n'est pas suffisant.

4.3.3 Estimation de la performance

En pratique, mesurer directement la performance pour tout intervalle de temps $]t_1, t_2]$ n'est pas possible car ceci nécessite de mesurer le temps après chaque candidat, c'est-à-dire de faire un appel à la fonction *clock()*. Ceci ajoute une opération supplémentaire à exécuter pour l'énumérateur, ce qui le ralentit. La Figure 4.3 illustre le processus de cassage de mots de passe avec ces appels supplémentaires m' et m''. Plus l'énumérateur est rapide dans la génération des candidats, plus il est impacté par les appels à *clock()*. Il aurait été possible de contourner ce problème en ne mesurant le temps qu'après un nombre fixe de candidats N, ne permettant plus de calculer la performance pour tout $]t_1, t_2]$. Cependant, nous avons choisi une manière plus convenable de calculer la performance, en séparant les mesures du taux de succès et de la fréquence, et en les mesurant pour des petites périodes afin d'en déduire une estimation de la performance. Ces mesures sont prises pour chaque intervalle d'une seconde :

$$P(j, j+1) = F(j, j+1)S(j, j+1)$$

où j est un instant et $j+1$ cet instant plus une seconde.

Le fait de mesurer séparément la fréquence et le taux de succès offre plusieurs avantages. Tout d'abord, les mesures n'ont pas besoin d'être effectuées en même temps, ce qui évite les interférences dans les mesures de temps. Deuxièmement, pour mesurer le taux de succès, il devient possible d'utiliser des astuces (lorsque disponibles) qui évitent de lancer l'énumérateur pour prendre les mesures, ce qui fait gagner du temps. Par exemple, pour la force brute, il est possible et efficace de calculer quand un mot donné sera énuméré en fonction de l'alphabet considéré et de la taille du mot (détails dans la section 1.3.2). Pour JtR-Markov, il est possible de calculer une approximation du rang d'énumération d'un mot de manière efficace (détails dans la section 1.4.2). Grâce à ces astuces, plutôt que de lancer l'énumération et attendre que l'algorithme trouve les mots de la base de données, il est possible

pour chaque mot de la base de données, de calculer son rang d'énumération. Ceci n'est cependant pas possible pour OMEN et PCFG, qui demandent d'être lancés pour mesurer leur taux de succès. Le troisième avantage est le fait que la mesure du taux de succès n'est pas influencée par les performances de la machine, ni perturbée par d'éventuels autres processus qui utiliseraient toutes les ressources. Si nous avions mesuré directement la performance, il aurait fallu être davantage vigilants sur le fait que les mesures n'aient pas été perturbées. Dans notre cas, seule la mesure de la fréquence est influencée par une surcharge de la machine, mais cela pose moins de difficultés que pour la performance.

D'abord, comme mentionné précédemment, la mesure de $c_g(t)$ a une influence sur sa valeur. Il est donc nécessaire d'effectuer le moins de mesures possibles tout en gardant une précision suffisante. Nous avons observé que $c_g(t)$ ne varie pas beaucoup entre j et $j+1$, sa valeur est donc supposée constante pour cette période. Notons c_g sa valeur, et mettons-lui comme valeur la moyenne de $c_g(t)$. Deuxièmement, c_h peut être considéré comme constant pour une fonction de hachage donnée, puisque la taille des mots de passe est presque toujours inférieure à la taille de l'entrée de la fonction de compression utilisée dans la fonction de hachage. Autrement dit, les données d'entrées sont bien souvent trop courtes pour faire varier significativement le calcul de l'empreinte.

Soit $c = c_g + c_h$ dans l'intervalle $]j, j+1]$, alors $F(j, j+1) = \frac{1}{c}$ est la fréquence de l'énumérateur pour cette période.

Considérons maintenant une période de k secondes $]t_1, t_2 = t_1 + k]$, on a

$$P(t_1, t_2) = \sum_{l=0}^{k-1} P(t_1 + l, t_1 + l + 1)$$

que l'on appelle la performance cumulée, à la manière des travaux de la littérature (sauf que le temps n'était pas pris en compte), qui utilisaient la métrique "number of guesses (NoG)", appelée fonction de répartition (Cumulative Distribution Function (CDF)).

4.4 Mesure des performances

Le calcul des performances n'a jamais été considéré, bien que le coût en temps de génération des candidats ait rapidement été mentionné dans un article de la littérature [48]. En particulier, il n'a jamais été mentionné que le taux de succès S et la performance F doivent être mesurés séparément si l'on veut pouvoir effectuer les mesures de manière précise et en un temps raisonnable. Une estimation de ces deux valeurs est suffisante pour avoir une bonne estimation de la performance. De plus, cela permet d'étudier ces deux propriétés de manière indépendante.

Néanmoins, comme mentionné dans [48], puisque certains énumérateurs sont très rapides dans leur énumération, mesurer $S(j, j+1)$ et $F(j, j+1)$ en pratique est un challenge technique. Les mesures effectuées sont maintenant présentées. Tout d'abord, aucune implantation d'énumérateur ne propose de mesurer le temps entre chaque candidat. L'implantation d'OMEN permet de mesurer le taux de succès

au fil des candidats générés. Pour les autres énumérateurs, il est nécessaire de prendre des mesures adaptées en fonction de l'implantation. Puisque le temps n'a pas d'importance lors de la mesure du taux de succès, il est possible d'utiliser des techniques différentes en fonction des énumérateurs.

Nombre de candidats générés pour une période. Les étapes (*ii*) et (*iii*) sont supposées constantes pour une fonction de hachage donnée, seule l'étape (*i*) a donc besoin d'être mesurée. Pour les mêmes raisons évoquées dans la section précédente, mesurer le temps entre chaque candidat jusqu'à atteindre une seconde est un procédé trop coûteux, surtout pour les énumérateurs très rapides. À la place, une estimation du nombre de candidats générés en une seconde est effectuée, en mesurant le temps nécessaire pour générer N candidats, on en déduit le nombre de candidats générés en une seconde. Par exemple, en moyenne PCFG génère environ 8×10^4 candidats par seconde, OMEN environ 10^6, JtR-Markov environ 2×10^7 et la force brute environ 1.6×10^9 (ces temps ne sont pas constants durant l'énumération, mais servent ici à avoir une idée de leurs valeurs). Afin de réduire l'erreur commise, on fait varier N jusqu'à ce que le temps moyen pour générer N candidats soit proche d'un dixième de seconde au début de l'énumération, afin d'avoir à la fois une bonne estimation de $c(t)$ et un nombre acceptable de nombres flottants à stocker, correspondant à chaque valeur de temps. On se ramène ensuite à 1 seconde avec un produit en croix. Les valeurs de N déterminées sont visibles dans la Figure 4.4.

Énumérateur	force-brute	JtR-Markov	OMEN	PCFG
N	10^9	4×10^6	10^5	10^4

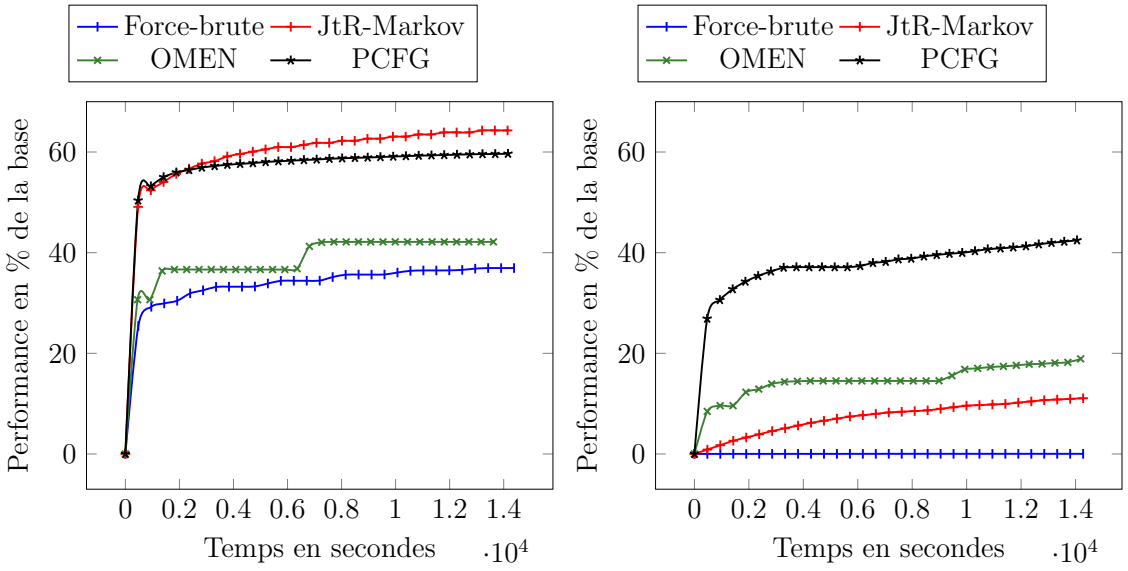
FIGURE 4.4 – valeur de N pour estimer la fréquence de chaque énumérateur

À l'issu de cette mesure, on obtient une liste de nombres flottants $T = [T_1, T_2, ..., T_k]$ où chaque T_i représente le temps nécessaire pour générer N candidats.

Taux de succès. Pour mesurer le taux de succès, il est nécessaire d'avoir pour chaque candidat c_i, son rang r_i dans l'énumération et son gain g^i (combien de fois il apparaît dans la base de données). Il existe deux manières de mesurer le taux de succès : en exécutant l'algorithme et en comptant les occurrences dans la base de données, ou à rebours en calculant pour chaque mot de passe de la base son rang dans l'énumération. La première méthode s'applique quelque soit l'énumérateur. Cependant, cela demande de les exécuter, ce qui prend du temps surtout si l'on souhaite faire cette mesure pour un grand nombre de candidats. La deuxième méthode n'est applicable que pour deux énumérateurs : JtR-Markov et la force-brute puisque le rang d'un mot est prédictible et facilement calculable grâce à la fonction de rang $R(m)$ tel que montré dans les sections 1.3.2 et 1.4.2.

À l'issu de cette mesure, on obtient une liste $SR = [(r_1, g^1), ..., (r_l, g^l)]$ de couples (rang, gain) pour chaque mot de passe de la base de données attaquée.

Calcul de la performance. Une fois que les taux de succès et les fréquences ont été mesurés, il est possible de calculer la performance en les appariant de la manière suivante : pour chaque $i \in \{0, ..., k\}$, on calcule la somme des gains des candidats de rang r tel que $r_{iN} \leq r < r_{(i+1)N}$.



(a) avec SHA-1 (b) avec bcrypt

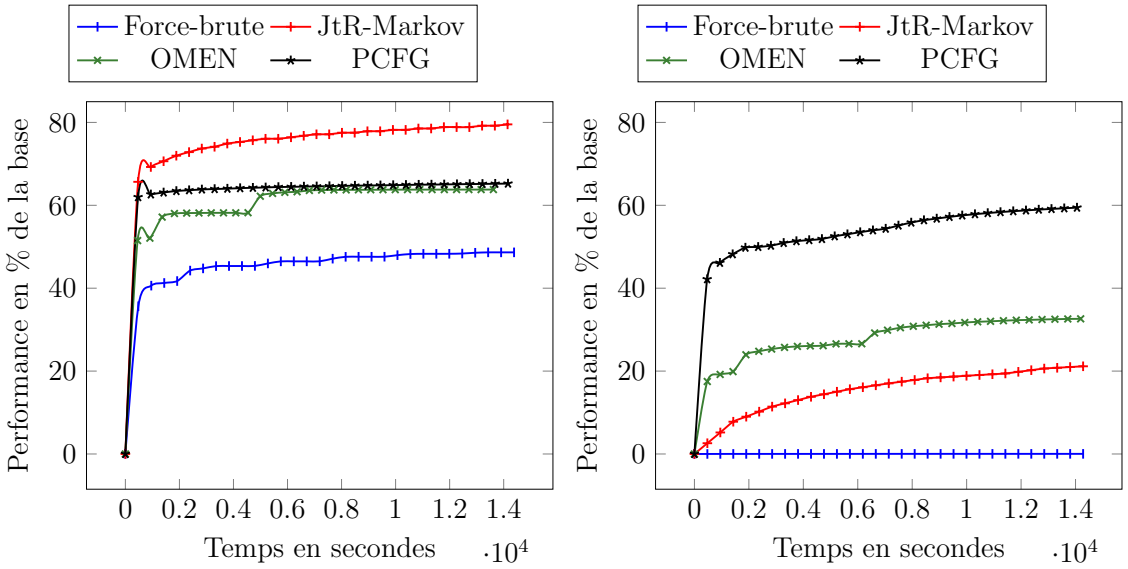
FIGURE 4.5 – Attaque sur LinkedIn

Courbe. La valeur du point correspondant a pour abscisse $\sum_{j=0}^{i} T_j$ (la somme des temps nécessaires pour générer les iN candidats précédents), et pour ordonnée P_i. On affiche donc le point $(\sum_{j=0}^{i} T_j, P_i)$.

4.5 Résultats expérimentaux

Dans cette section sont présentés les résultats du calcul des performances pour quatre énumérateurs : la force brute, le mode Markov de John The Ripper, OMEN et l'énumérateur basé sur PCFG, sur les bases de données LinkedIn et Rockyou, en simulant deux fonctions de hachage, SHA-1 et bcrypt (avec un coût de 10). Plutôt que de tracer les courbes de performances telles quelles, on trace le pourcentage de la base de donnée trouvé depuis le début. Ainsi, cela permet de se comparer aux travaux de la littérature mais aussi de comparer les graphiques entre eux puisque les bases de données ne sont pas de même taille. Les mesures de vitesses des fonctions de hachage proviennent de [19], où pour SHA-1, $1/c_h \approx 12.5 \times 10^9$, et pour bcrypt avec un coût de 10, la valeur par défaut de la fonction *password_hash()* de PHP, $1/c_h \approx 700$. Les mesures dans Hashcat sont effectuées sur bcrypt avec un coût de 5, ce qui signifie 2^5 tours de la fonction de dérivation de clé interne. Ainsi, c_h pour un coût de 10 est 2^5 plus petit, donnant $23 \times 10^3/2^5 \approx 700$.

Sur la Figure 4.5a montrant les performances des énumérateurs en attaquant LinkedIn protégée par SHA-1, JtR-Markov et PCFG cassent environ 60% des mots de passe après quatre heures, tandis qu'OMEN ne casse qu'environ 43% et la force brute 36%. La force brute, malgré la naïveté de la méthode, montre tout de même de bons résultats. PCFG est étonnamment bon puisqu'il exploite le fait qu'un grand nombre de mots de passe partagent la même structure dans cette base de données :



(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.6 – Attaque sur Rockyou

37% des mots de passe partagent les 5 structures les plus fréquentes. Cependant, en simulant bcrypt (Fig. 4.5b), les performances des énumérateurs sont chamboulées. À cause de la faible qualité des candidats de la force brute, le processus passe son temps à hacher des candidats n'ayant que peu de succès. Après quatre heures, la force brute a cassé moins de 1% de la base de donnée, tandis que PCFG continue d'être efficace en cassant 43% de la base. OMEN remonte dans le classement en cassant plus de 18% de la base. Finalement, JtR-Markov est moins bon qu'OMEN, en ne trouvant que 11% des mots de passe après quatre heures.

En attaquant Rockyou protégée par SHA-1 (Fig. 4.6a), JtR-Markov casse 15% des mots de plus que sur LinkedIn après quatre heures, où OMEN casse 23% de plus que sur LinkedIn, et la force brute 12% de plus. Cependant, PCFG conserve des performances similaires à celles sur LinkedIn avec du SHA-1. Sur Rockyou avec du bcrypt (Fig. 4.6b), les résultats sont similaires à ceux contre LinkedIn avec du bcrypt : même si les énumérateurs cassent proportionnellement moins de mots de passe, l'ordre des performances est conservé. Les "marches" visibles sur les courbes d'OMEN est dû au fait que l'algorithme génère, par ensemble, des candidats de même longueur, et que les candidats dans ces ensembles sont générés par probabilité décroissante. Quand OMEN change la longueur des candidats, il casse soudainement beaucoup de mots de passe.

4.5.1 Bases privées des mots de passe les plus faibles

Mot de passe faible : dans les expériences menées ici, nous désignons un mot de passe comme faible s'il fait partie des premiers mots de passe attaqués par un adversaire. Ceci regroupe les mots de passe les plus trivialement cassables. Dans le premier scénario, les mots de passe faibles sont ceux trouvés par une force brute

pendant une heure environ. Dans le deuxième scénario, ce sont ceux qui ne satisfont pas une politique de sécurité donnée.

4.5.1.1 Règles des professionnels du cassage de mots de passe

Dans ce scénario, on attaque des sous-ensembles des bases de données où les mots de passe les plus faibles ont été retirés. Les mots de passe de taille 6 ou moins, ceux tout en minuscules de taille 7 ou 8, en majuscules de taille 7 ou 8, et ceux ne contenant que des chiffres de taille 7 à 12 sont retirés des bases de données. Ces mots de passe faibles sont généralement attaqués en premier par les professionnels du cassage de mots de passe [18]. Dans cette partie comme dans la suivante, l'étape d'apprentissage est exécutée à nouveau sur la moitié de l'ensemble filtré. Par exemple, s'il n'y a pas de mot de longueur inférieure à 8, l'énumérateur ne testera pas de mots de longueur inférieure à 8.

Si l'on compare à l'attaque sur la base entière de LinkedIn (Figure 4.5a), PCFG est l'énumérateur qui souffre le moins du manque de mots de passe faibles (Figure 4.7a), car il perd moins de 5% de performance, tandis que JtR-Markov et OMEN en perdent quasiment 20%. La force brute devient beaucoup moins intéressante car elle attaque essentiellement des mots de passe très faibles au début. Quand bcrypt est utilisé (Figure 4.7b), même si l'on peut faire les mêmes remarques pour la force brute, OMEN et PCFG, on constate que JtR-Markov ne casse quasiment pas de mots de passe, et devient quasiment aussi peu efficace que la force brute.

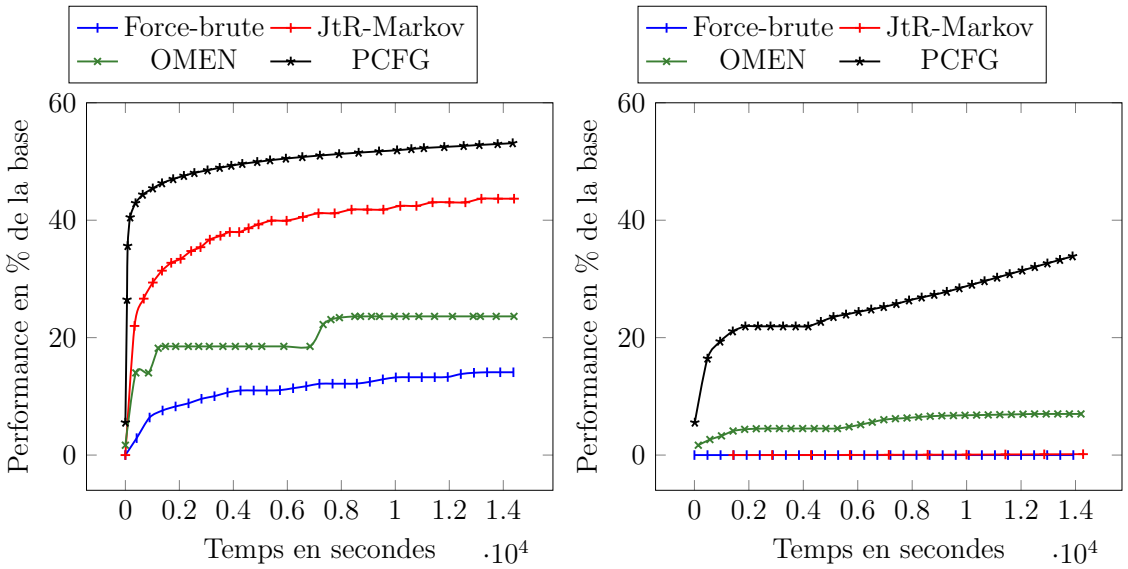
En attaquant la base de données Rockyou avec SHA-1 sans les mots de passe les plus faibles (Figure 4.8a), le phénomène est similaire à l'attaque sur LinkedIn, c'est-à-dire que PCFG est peu impacté, tandis que les trois autres énumérateurs perdent environ 20% de performance chacun. Quand bcrypt est utilisé, chaque énumérateur perd également environ 20% de performance sauf OMEN, qui lui n'en perd que 10. Ceci est dû au fait qu'OMEN s'adapte à la base de données attaquée en réordonnant les ensembles de mots qu'il parcourt.

Cette expérience montre que même si l'on retire les mots de passe les plus faibles des bases de données, les performances des énumérateurs restent satisfaisantes. Il est donc évident que retirer les mots de passe les plus faibles, tel que défini lors de cette expérience, n'est pas suffisant à faire baisser les performances des énumérateurs étudiés.

4.5.1.2 Règles heuristiques étudiées par les académiques

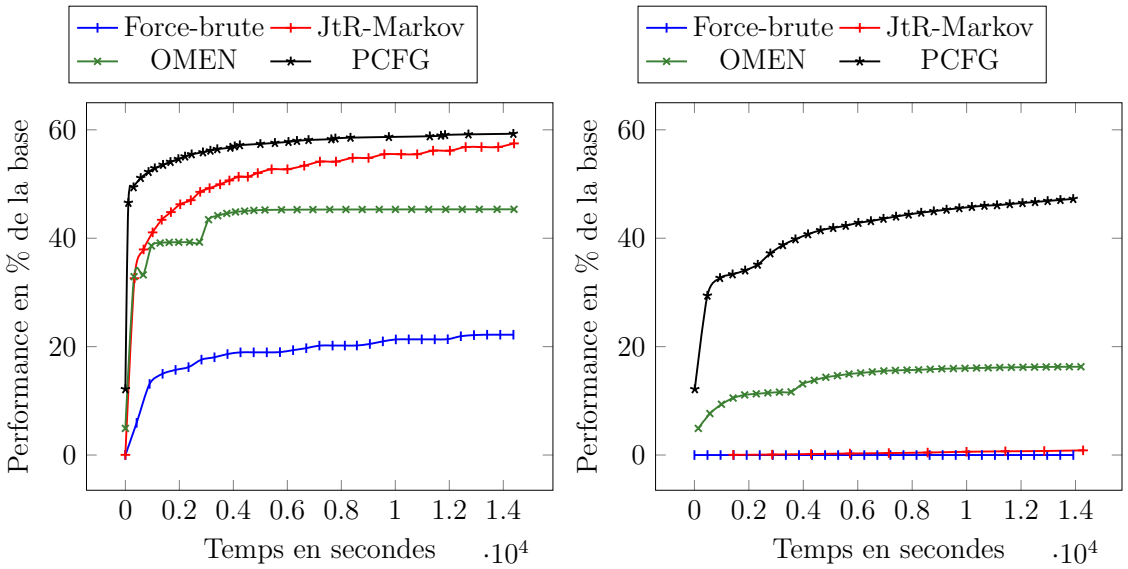
Dans cette partie, nous étudions les performances des énumérateurs sur des sous-ensembles de la base LinkedIn, définis selon 4 politiques de composition telles que présentées dans l'article [48]. Les performances peuvent en effet varier selon les mots présents dans la base attaquée, comme le montrent les résultats de l'expérience précédente, où PCFG était meilleur quand les mots de passe faibles étaient retirés de la base. Les 4 sous-ensembles considérés dans cette expériences sont :

— **basic** : les mots de passe conservés contiennent au moins 8 caractères



(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.7 – Attaque sur LinkedIn sans les mots de passe faibles



(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.8 – Attaque sur Rockyou sans les mots de passe faibles

— **complex** : les mots de passe contiennent au moins 8 caractères dont un de chaque classe (minuscule, majuscule, chiffres, symboles)

— **longbasic** : les mots de passe contiennent plus de 16 caractères

— **longcomplex** : les mots de passe contiennent plus de 12 caractères d'au moins 3 classes différentes

Ces politiques de composition, bien qu'arbitraires, représentent convenablement celles utilisées en pratique dans les services en ligne.

Il faut noter que le filtre **basic** est très proche du début de l'attaque par les professionnels tel qu'effectuée dans l'expérience précédente. On a donc une forte ressemblance entre les graphiques 4.7 et 4.9.

Lorsque la politique **basic** est appliquée sur LinkedIn (Figure 4.9a), la force brute est beaucoup moins performante car l'espace des mots de taille 8 et plus est plus long à explorer. Ceci est valable aussi pour les trois autres politiques de composition appliquées à LinkedIn. Cependant, PCFG est toujours aussi performant et trouve 60% de la base en 14400 secondes, autant que sur la base originale, tandis que JtR-Markov en trouve 50%, soit plus de 10% en moins que sur la base originale. OMEN quant à lui a des performances similaires, même si la courbe de sa performance est moins pentue que sur la base d'origine.

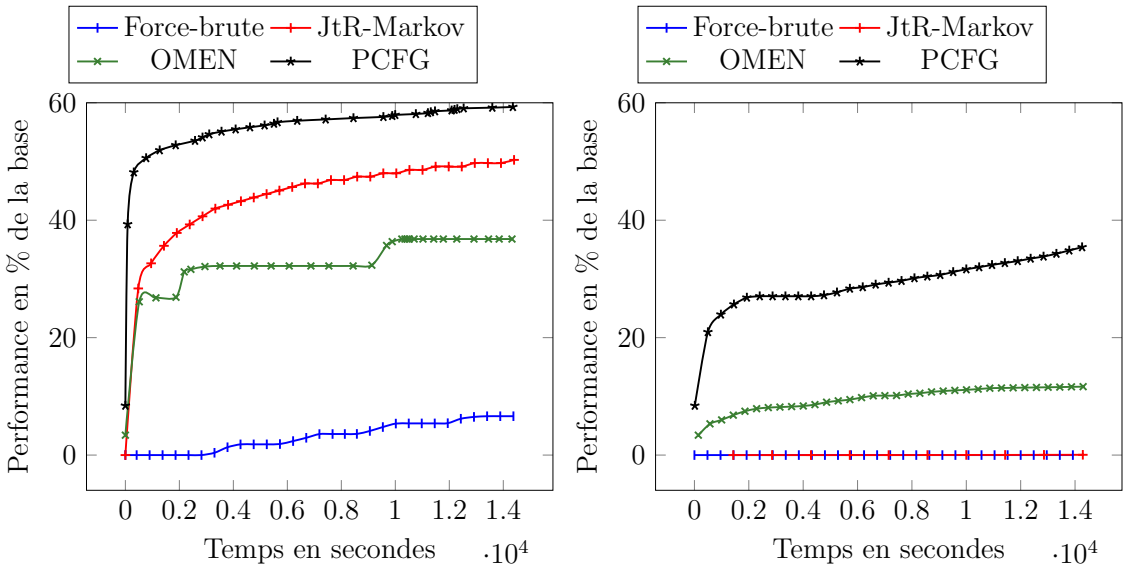
Concernant l'attaque de la base LinkedIn filtrée avec la politique **complex** (Figure 4.10a), les trois énumérateurs JtR-Markov, OMEN et PCFG cassent quasiment autant de mots de passe après 4 heures, cependant leur progression est différente. Néanmoins, les performances des énumérateurs sur l'attaque de la base avec la politique **longbasic** (Figure 4.11a) sont beaucoup moins bonnes, excepté pour PCFG. JtR-Markov et la force brute ne trouvent quasiment pas de mots de passe en 4 heures, et OMEN en trouve moins de 10%, alors que PCFG trouve environ 30%, ce qui montre encore une fois que PCFG est meilleur lorsque les mots de passe les moins robustes sont retirés.

Sur la base LinkedIn avec la politique **longcomplex** (Figure 4.12a), l'écart entre PCFG et OMEN est moins important qu'avec la politique **longbasic**. En revanche, JtR-Markov trouve 1% de la base de données, comparée à 0% sur LinkedIn avec la politique **longbasic**, dû au fait que les mots de passe ici sont en moyenne plus courts.

On remarque que par ces différentes expériences, les performances des énumérateurs sont très dépendantes des bases de données attaquées. Certains énumérateurs sont meilleurs lorsque beaucoup de mots de passe faibles sont présents, comme c'est le cas de JtR-Markov. Cependant, dès que l'on retire une partie des mots de passe faibles, on constate que PCFG devient nettement meilleur que les autres énumérateurs, même si avec certaines politiques de composition les performances sont sensiblement les mêmes.

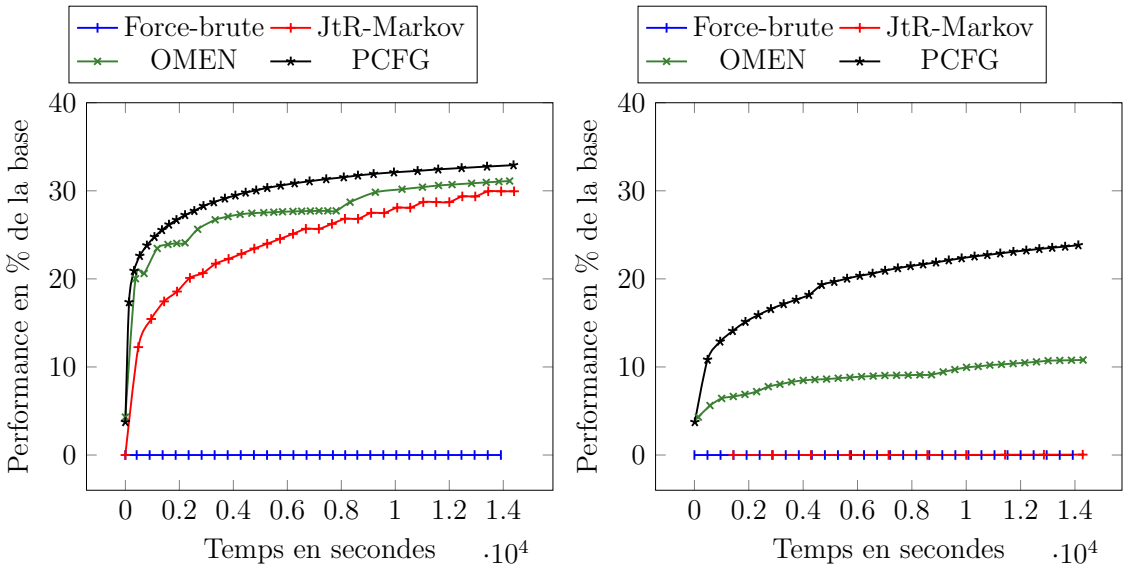
4.6 Conclusion

Les expériences menées dans ce chapitre montrent que la performance du processus de cassage des mots de passe est dépendant non pas seulement de son taux de succès,



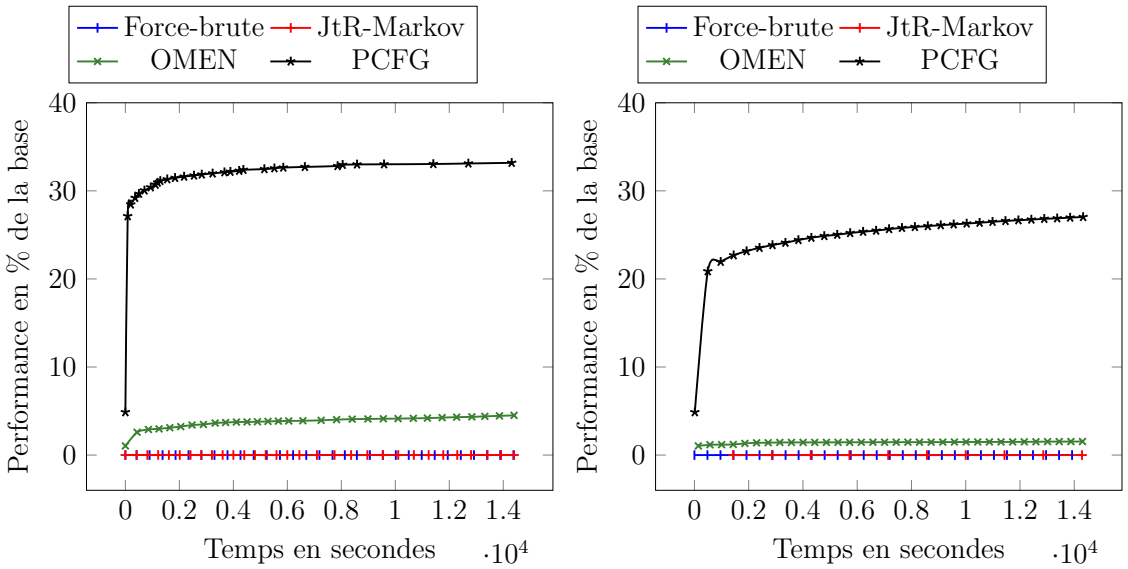
(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.9 – Attaque sur LinkedIn + filtre basic



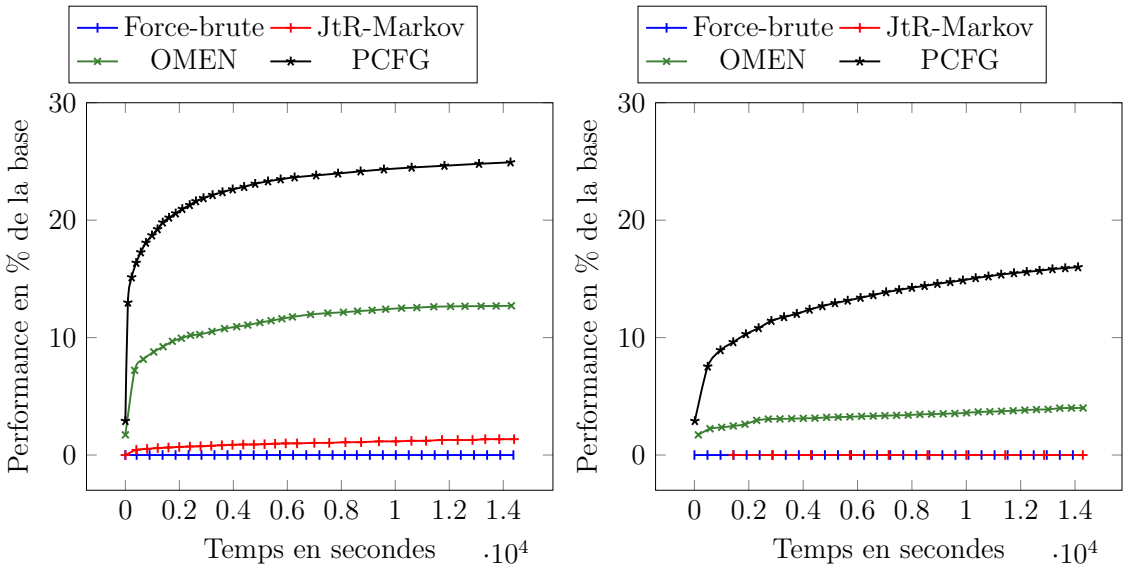
(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.10 – Attaque sur LinkedIn + filtre complex



(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.11 – Attaque sur LinkedIn + filtre longbasic



(a) avec SHA-1 (b) avec bcrypt

FIGURE 4.12 – Attaque sur LinkedIn + filtre longcomplex

mais également de la vitesse d'énumération des candidats et de la vitesse de la fonction de hachage. En effet, si l'on utilise une fonction rapide telle que SHA-1, on permet aux énumérateurs les plus rapides de casser beaucoup de mots de passe, même si les candidats générés sont de mauvaise qualité. Par exemple la force brute reste une méthode viable pour casser des mots de passe sur LinkedIn avec SHA-1 car elle est quasiment aussi performante que OMEN. Si cette force brute était implémentée sur carte graphique, elle serait probablement meilleur que OMEN dans ces conditions. En revanche, utiliser bcrypt pour protéger les mots de passe permet de faire baisser fortement les énumérateurs rapides tels que JtR-Markov et la force brute. Cependant, cela n'empêche pas les énumérateurs probabilistes tels que PCFG et OMEN d'être efficaces. Les expériences montrent également que PCFG est le meilleur énumérateur dans quasiment toutes les conditions testées, même lorsque la fonction de hachage est rapide. PCFG est notamment largement meilleur que les autres énumérateurs lorsque l'on retire les mots de passe faibles des bases de données. Ceci indique que PCFG a une meilleure capacité d'adaptation aux spécificités des bases de données que ses concurrents, l'apprentissage étant refait à chaque fois.

Les expériences sur les bases de données filtrées où l'on retire les mots de passe les plus faibles montrent que les filtres étudiés ne sont pas suffisants pour contrer les attaques utilisant des énumérateurs probabilistes. Il est donc nécessaire aujourd'hui d'utiliser des métriques de robustesse plus adaptées à ces énumérateurs. Le chapitre suivant traite de ce problème, à savoir à quel point les métriques heuristiques de robustesse sont pertinentes pour protéger les mots de passe des énumérateurs probabilistes.

Chapitre 5

Efficacité des politiques heuristiques contre les modèles probabilistes

Dans ce chapitre, on évalue l'efficacité des mesures de robustesse heuristiques pour se protéger des attaques utilisant des modèles probabilistes. Les mesures de robustesse heuristiques considérées dans ce chapitre sont celles qui imposent des règles obligatoires que doivent satisfaire les mots de passe pour être acceptés. En utilisant les outils formels et techniques de la première contribution, on s'intéresse ici à étudier ce qui fait qu'une base de données est résistante face aux différents énumérateurs probabilistes, car cela permet de mettre en avant la robustesse de cette base mais aussi la robustesse des mots qui la composent. Pour cela, nous étudions les performances des énumérateurs de manière locale, sur une courte période de temps, de manière à percevoir l'évolution de leur comportement au cours du temps. Même si les expériences du chapitre 4 montrent que ***complex*** *et* ***longcomplex*** *ne permettent pas de réduire suffisamment les performances des énumérateurs, les expériences menées dans ce chapitre permettent d'avoir un angle d'observation différent. Ici, l'idée n'est pas de comparer les énumérateurs entre eux ni de comparer les métriques de robustesse entre elles, mais plutôt d'étudier l'efficacité de chaque métrique de robustesse pour protéger une base de données contre chaque énumérateur.*

Sommaire

5.1 Introduction

L'hypothèse que l'on souhaite tester dans ce chapitre est la suivante : les métriques heuristiques de robustesse des mots de passe sont inefficaces pour protéger les mots de passe contre les énumérateurs basés sur des modèles probabilistes. Pour réaliser ce test, on mesure la performance locale des attaques utilisant un énumérateur probabiliste pour casser les mots de passe d'une base de données filtrées avec 3 différentes métriques heuristiques : *complex*, *longcomplex* et *ZXCVBN*. La mesure de la performance est effectuée telle que décrite dans le chapitre 4. Puisque *ZXCVBN* est paramétrable, on teste également l'hypothèse sur les bases filtrées avec les différents scores proposés par *ZXCVBN*. Puisque l'on souhaite savoir si les métriques heuristiques sont efficaces pour rendre les mots de passe plus robustes face aux énumérateurs probabilistes, il est peu pertinent de simuler l'usage d'une fonction de hachage rapide car ceci constitue déjà un avantage pour l'attaquant. De plus, les conclusions n'auraient pas permises de mesurer l'apport en sécurité apporté par les métriques heuristiques par rapport à l'utilisation d'une fonction de hachage adaptée au stockage des mots de passe. C'est pourquoi, dans ce chapitre, nous simulons l'usage de bcrypt pour protéger les mots de passe en base de données. Conduire une étude de la performance de ces attaques permet de montrer de quoi l'attaquant est capable. Ceci permet également de donner des conseils pertinents pour choisir des politiques de sécurité des mots de passe en fonction de la sécurité et l'utilisabilité de chaque méthode. Ce travail peut être observé de deux points de vue différents. Le premier permet de s'intéresser à l'efficacité des énumérateurs probabilistes face aux bases filtrées en faisant varier la base des mots de passe. Ce point de vue permet de se focaliser sur les énumérateurs. Le deuxième point de vue permet d'étudier la robustesse d'une base ou d'une base filtrée face aux différents énumérateurs probabilistes afin de déterminer l'attaque la plus performante. De ce point de vue, on s'intéresse davantage à étudier ce qui fait qu'une base de données est résistante face à ces énumérateurs probabilistes, car cela permet de mettre en avant la robustesse de cette base mais aussi la robustesse des mots de passe qui la composent.

Nous montrons dans ce chapitre que même les métriques heuristiques les plus élaborées telles que *ZXCVBN* associées avec une fonction de hachage lente ne permettent pas de protéger efficacement les mots de passe face aux énumérateurs probabilistes, car ils restent performants même après avoir retiré les mots de passe les plus triviaux. Cependant, les 3 métriques testées sont tout de même efficaces pour protéger des attaques les plus naïves telles que le mode Markov de John The Ripper et la force brute. Si l'on part dans l'optique que les fonctions de hachage lentes grandissent en popularité, l'usage des énumérateurs probabilistes devrait se généraliser car ce sont les seules méthodes efficaces contre ces fonctions, comme montré dans le chapitre 4, ce qui implique de remplacer ces métriques heuristiques avec des méthodes nouvelles qui protègent de ces énumérateurs.

Ce chapitre est organisé de la manière suivante : premièrement, nous présentons le contexte du travail en section 5.2, deuxièmement la manière dont sont filtrées les bases de données est présentée en section 5.3, les détails des expériences sont présentés en section 5.4, et enfin les résultats sont exposés en section 5.5. Nous

conclurons en section 5.6.

5.2 Contexte

5.2.1 Bases et énumérateurs étudiés

Les 3 bases de données étudiées dans ce chapitre sont **Rockyou**, **LinkedIn** et **Myspace**. Les énumérateurs probabilistes étudiés dans ce chapitre sont la PCFG de Weir, OMEN et le mode Markov de John The Ripper (JtR-Markov). Simuler une attaque utilisant la force brute aurait été naturel afin de la comparer aux autres énumérateurs, cependant le fait d'utiliser bcrypt avec les métriques heuristiques ne permet pas à la force brute d'être efficace, entraînant une performance trop faible. C'est pourquoi les performances de la force brute ne sont pas montrées.

De la même manière que dans le chapitre précédent, l'attaque d'une base de données ou d'une base de données filtrée s'effectue en deux étapes : l'apprentissage est exécuté sur une moitié aléatoire de la base et l'évaluation de l'énumération s'effectue sur la deuxième moitié. L'étape d'apprentissage est donc répétée pour chaque base de données attaquée afin que les énumérateurs puissent s'adapter aux spécificités de chaque base.

5.2.2 Utilisabilité

Une politique de sécurité des mots de passe est un ensemble de règles que doit satisfaire un mot de passe pour être accepté dans le système. Un aspect important lors de l'utilisation d'une politique de sécurité des mots de passe est l'utilisabilité. Cette notion définit à quel point une politique de sécurité est acceptée par les utilisateurs finaux. Pour une politique de sécurité des mots de passe, elle est souvent mesurée en le nombre de fois où un utilisateur doit modifier son mot de passe afin qu'il soit accepté par cette politique. Une politique très utilisable accepte facilement les mots de passe. La difficulté principale lorsque l'on conçoit des politiques de sécurité des mots de passe est de gérer le compromis entre la sécurité et l'utilisabilité qu'elles offrent, même si ces deux propriétés ne sont pas incompatibles.

5.2.3 Fonctions de hachage

Il est important également de bien choisir la fonction de hachage pour le stockage des mots de passe. Même si dans les fuites de données récentes, les fonctions rapides restent majoritairement utilisées (voir Figure 1.1), elles ne sont pas recommandées car elles donnent un avantage à l'attaquant. Il faut donc utiliser des fonctions lentes. Pour les mêmes raisons qu'au chapitre 4, nous choisissons d'utiliser bcrypt comme représentante des fonctions de hachage lentes.

	LinkedIn	Rockyou	Myspace
Origine	175087762	32603388	308875476
longcomplex	11719618 (6.7%)	1436604 (4.4%)	0
complex	13452789 (7.7%)	479432 (1.5%)	0
1	161142769 (82%)	27993172 (85.8%)	296253876 (95.9%)
2	99698258 (56.9%)	13175372 (40.4%)	131164838 (42.4%)
3	54340823 (31%)	5573554 (17%)	45188326 (14.6%)
4	22778267 (13%)	1247387 (3.8%)	2312808 (0.7%)

FIGURE 5.1 – Taille des ensembles issus des processus de filtrage (proportion par rapport à la base de données originale)

5.3 Filtres sur les bases de données

Afin de mesurer les performances des énumérateurs contre des mots de passe de différentes robustesses, nous simulons deux scénarios utilisés en pratique pour renforcer la robustesse des mots de passe. Le premier est l'usage des heuristiques basiques *complex* et *longcomplex* : les mots de passe de la base de données sont filtrés en fonction de règles simples telles que la longueur du mot ou les caractères qui le composent. Lorsque les mots de passe sont filtrés avec *complex*, seuls ceux de taille 8 ou plus et ayant au moins un caractère de chacune des 4 classes sont conservés. Pour *longcomplex*, les mots de passe de taille 12 ou plus et ayant au moins 3 classes de caractères différentes sont conservés. Myspace, de part la manière dont les mots de passe étaient stockés (voir 1.1.6), ne peut pas être filtrée avec ces deux métriques basiques car tous les mots sont de taille 10 ou moins (*longcomplex* impossible) et tout en minuscules (*complex* impossible). Le deuxième scénario est l'usage d'une métrique heuristique plus élaborée, *ZXCVBN* : un mot de passe est conservé si le score retourné par *ZXCVBN* est supérieur à un paramètre fixé par le service. Pour chaque base de données $D \in \{LinkedIn, Rockyou, Myspace\}$ et pour chaque score $s \in \{0, 1, 2, 3, 4\}$, nous filtrons D en retirant les mots de passe de score $< s$. Par exemple, si $s = 3$ sur LinkedIn, on retire les mots de passe de score 0, 1 et 2 et on conserve ceux de score 3 et 4. La table 5.1 décrit la taille de chaque ensemble issu de ces processus de filtrage.

Il faut noter que les filtres sont appliqués sur des bases de données existantes, ce qui introduit un biais car dans des conditions réelles d'application de ces métriques, les mots de passe rejetés peuvent être modifiés jusqu'à être acceptés.

5.4 Expériences

5.4.1 Mesure de la performance

Nous souhaitons comparer le comportement d'un énumérateur sur la base de données d'origine avec les versions filtrées de cette base en calculant le pourcentage de mots de passe trouvés au cours du temps. Pour cela, nous devons d'abord montrer

qu'au long de l'énumération, le pourcentage de mots de passe trouvés d'un sous-ensemble aléatoire SD par un énumérateur E est sensiblement le même que sur la base de données originale D, à condition que SD soit suffisamment grand. On peut montrer cela en effectuant le tirage aléatoire avec des essais de Bernoulli.

Sous-base aléatoire : Soit p la moyenne de la proportion de mots de passe dans SD (la rapport entre le cardinal de SD et celui de D). Chaque mot de passe de D est ajouté dans SD avec probabilité p. Puisque les essais sont indépendants, SD peut être construit durant l'énumération. Quand un mot de passe de D est trouvé, on effectue un essai de Bernoulli pour savoir si ce mot est ajouté dans SD. Les graphiques présentés dans la section suivante montrent la moyenne du pourcentage de mots de passe trouvés dans l'intervalle des 140 secondes qui précèdent le temps donné. Soit t (resp. s) le nombre de mots de passe trouvés dans D (resp. SD) dans un tel intervalle.

Proposition : la valeur moyenne de s est $t.p$, et la probabilité d'avoir un écart grand entre s et $t.p$ peut être borné par une borne de Chernoff.

Preuve : pour tout $\delta > 0$, on a $Pr(|t.s - t.p| > \delta.t.p) \leq e^{-\frac{2t.p.\delta^2}{3}}$.

Exemple d'application : avec la base de données LinkedIn et $p = 0.13$, si nous avons une moyenne de 0.1% des mots de passe trouvés dans un intervalle de 140s, nous avons $t.p = 0.001 \times 22778267 \times 0.13 = 2961$, ce qui permet d'assurer un pourcentage très similaire de SD dans le même intervalle. Ainsi, les graphiques resterons très similaires, sauf quand la moyenne des pourcentage devient très faible, mais cela ne change pas la progression de l'attaque. Nous nous attendons à des résultats différents du fait que les processus de filtrage des bases de données ne sont pas des tirages aléatoires.

5.4.2 Paramètres d'expérimentation

Dans nos expériences, nous avons choisi de tracer les courbes de la moyenne de la performance sur un intervalle de 140 secondes, ce qui représente 1% du temps total de l'attaque. On trace donc un point au plus toutes les 140 secondes représentant la moyenne de la performance sur les 140 secondes qui précèdent. Ce temps est pertinent car cela permet d'observer les phénomènes locaux tout en gardant des graphiques assez lisibles. L'avantage de tracer la moyenne de la performance sur des périodes courtes est que cela permet de comprendre les comportements différemment que dans le chapitre précédent. En effet, lorsque l'on trace les courbes de manière cumulative, c'est-à-dire avec le pourcentage de mots trouvés depuis le début de l'énumération, les variations locales sont camouflées par la progression globale. En traçant la moyenne sur des périodes courtes, on peut observer les hausses et les baisses de performances tout au long de l'énumération. On peut donc détecter les énumérateurs qui deviennent très performants après une certaine période de temps, et ceux qui deviennent très peu performants au bout d'un certain temps. La Figure 5.13 en est un bon exemple, où les performances sont parfois très faibles, et parfois augmentent d'un coup.

De plus, nous avons choisi de tracer les courbes sur quatre heures d'attaque, parce que nous avons observé que ce temps était suffisant pour représenter les performances

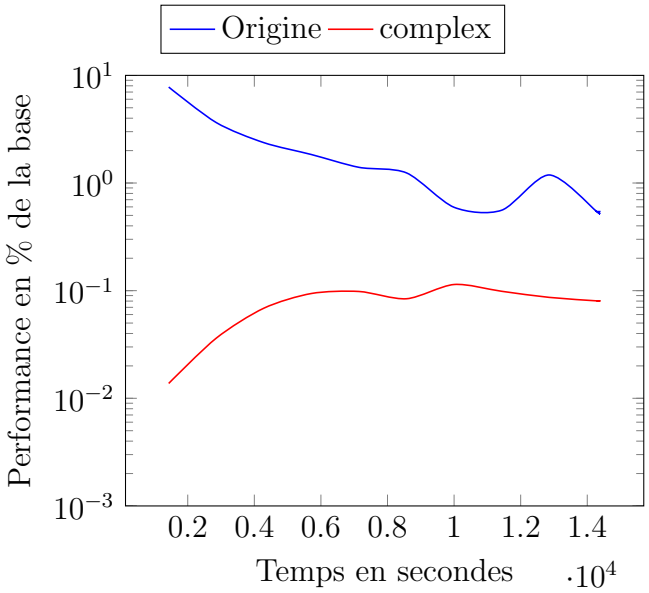


FIGURE 5.2 – Performance de JtR-Markov sur Rockyou avec et sans filtre complex

de chaque énumérateur. Au delà de quatre heures, l'ordre des courbes reste inchangé, c'est pourquoi nous avons tronqué les attaques pour simplifier les graphiques.

5.5 Résultats

Dans cette section, nous présentons les graphiques de la performance des énumérateurs sur les 3 bases de données avec chacun des filtres. L'axe des abscisses représente le temps depuis le début de l'attaque, et l'axe des ordonnées représente la moyenne du pourcentage trouvé de la base de données dans les 140 secondes qui précèdent. L'axe des abscisses est fixé sur l'intervalle $[0, 14400]$ et l'axe des ordonnées est fixé sur l'intervalle $[10^{-3}, 10^{1}]$ toujours pour des raisons de simplicité d'affichage.

5.5.1 Heuristiques basiques

Nous présentons maintenant les performances locales des énumérateurs probabilistes pour casser les mots de passe des bases de données avec et sans filtre.

5.5.1.1 Mode Markov de John The Ripper

Le mode Markov de John The Ripper, contrairement à PCFG et OMEN, est fortement impacté par l'application des filtres heuristiques basiques, comme le montrent les figures 5.2 et 5.3. Dans de tels scénarios, le filtre *longcomplex* permet de bloquer totalement l'attaque, tandis que le filtre *complex* ne permet de réduire que d'un facteur 10 la performance de l'attaque.

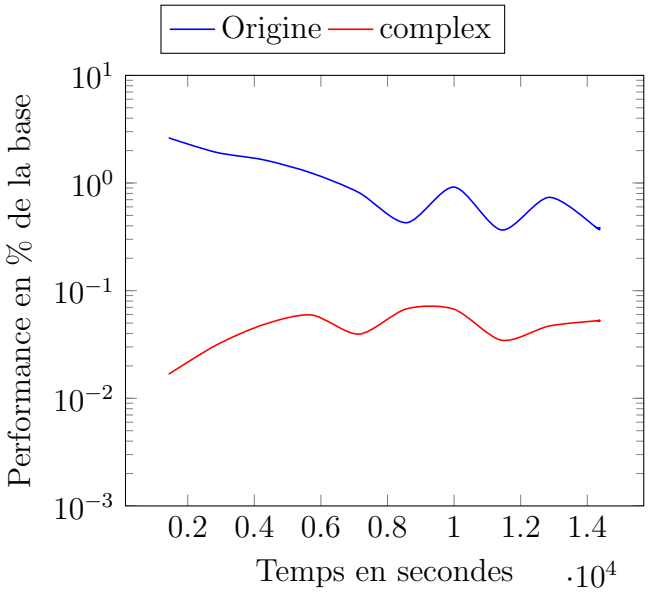


FIGURE 5.3 – Performance de JtR-Markov sur LinkedIn avec et sans filtre complex

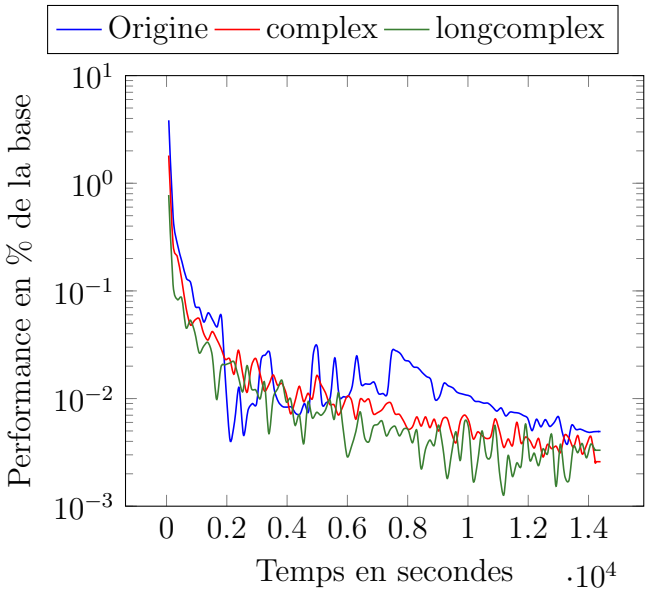


FIGURE 5.4 – Performance de PCFG sur Rockyou avec et sans filtres heuristiques basiques

5.5.1.2 PCFG

La figure 5.4 montre que globalement le filtre *complex* réduit les performances de PCFG sur Rockyou comparé à l'attaque sans filtre. Le filtre *longcomplex* est encore plus efficace. Cependant, les performances diffèrent de moins d'un ordre de grandeur, ce qui signifie que PCFG trouve avec ces filtres, plus de 10% de ce qu'il trouve sans filtre. La figure 5.5 révèle que PCFG est tout autant impacté sur LinkedIn que sur Rockyou. Cependant, entre 3000 et 6000 secondes, PCFG énumère

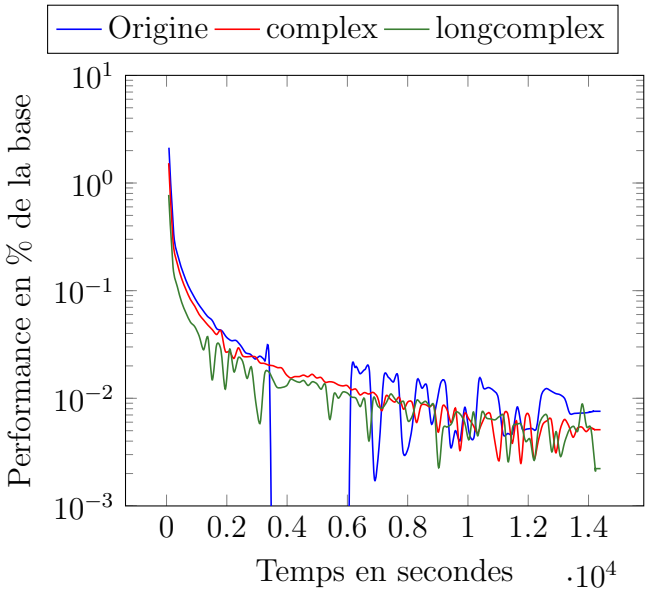


FIGURE 5.5 – Performance de PCFG sur LinkedIn avec et sans filtres heuristiques basiques

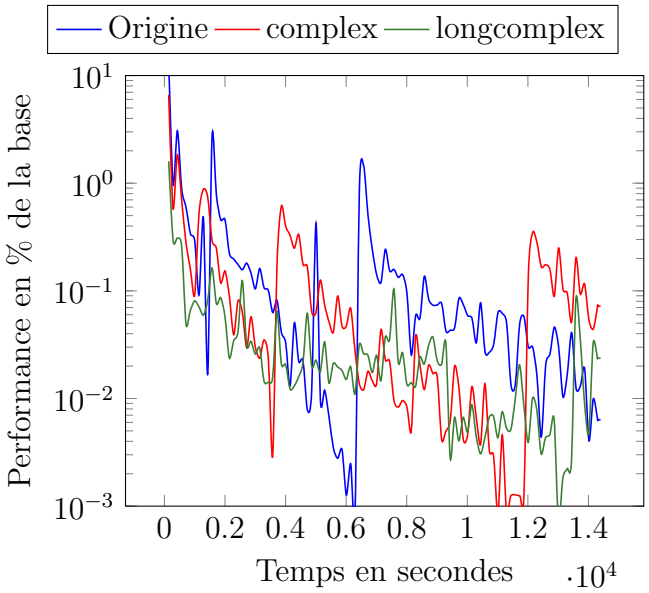


FIGURE 5.6 – Performance de OMEN sur Rockyou avec et sans filtres heuristiques basiques

les candidats de l'une de ses structures pour laquelle le taux de succès est proche de 0. Ce comportement est observable également sur la figure 5.13 avec le filtre ZXCVBN.

5.5.1.3 OMEN

OMEN est moins impacté par l'application de ces heuristiques basiques. C'est probablement grâce à sa stratégie adaptative. Comme on peut le voir sur la figure 5.6,

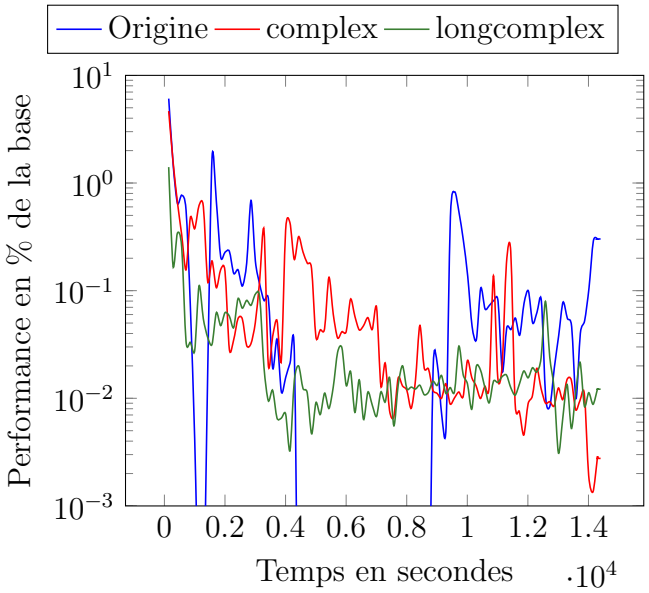


FIGURE 5.7 – Performance de OMEN sur LinkedIn avec et sans filtres heuristiques basiques

il n'y pas un seul meilleur filtre pour toute la durée de l'attaque. Ce phénomène peut également s'observer sur la base LinkedIn en figure 5.7, bien qu'entre 4000 et 9000 secondes, OMEN a une mauvaise performance sur la base d'origine. C'est probablement un effet pervers de sa stratégie adaptative : durant cette période, OMEN parcourt un très grand ensemble de mots de taille inférieure à 8, dont très peu sont des mots de passe. En effectuant l'apprentissage sur les bases filtrées, OMEN s'adapte et ne propose plus ces mots car ils sont trop courts pour passer les filtres, ce qui lui permet d'essayer d'autres mots de taille plus grande plus tôt dans l'énumération.

Avec ces observations, nous montrons que le filtre *longcomplex* est le meilleur des deux pour réduire les performances des énumérateurs. Ceci était déjà davantage visible sur les figures 4.10 et 4.12 du chapitre précédent. Cependant, même avec l'utilisation de bcrypt, ces deux heuristiques basiques n'ont pas un impact suffisamment significatif sur les performances d'OMEN et PCFG sur des durées d'attaque longues.

5.5.2 ZXCVBN

Chaque courbe représente la performance de l'énumérateur observé sur la base de données filtrée avec le score de ZXCVBN correspondant.

On observe que la plupart du temps, augmenter le score requis des mots de passe réduit la performance des attaques. Cependant, ceci n'a pas le même impact sur tous les énumérateurs.

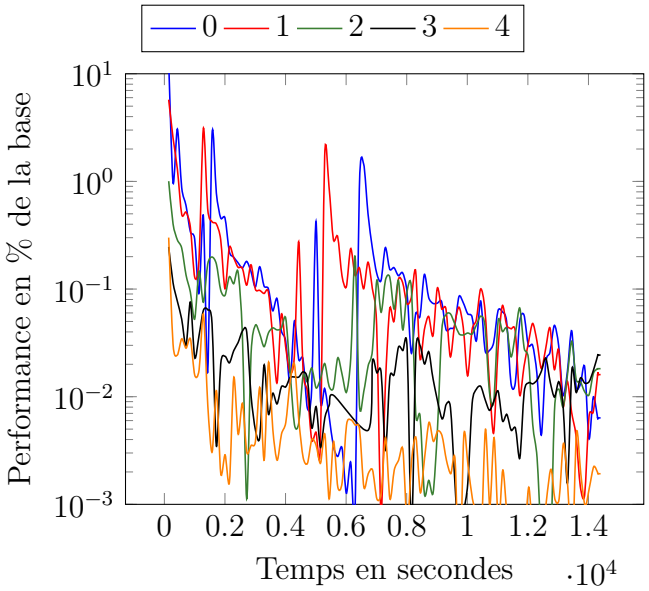


FIGURE 5.8 – Attaque de JtR-Markov sur Rockyou avec ZXCVBN

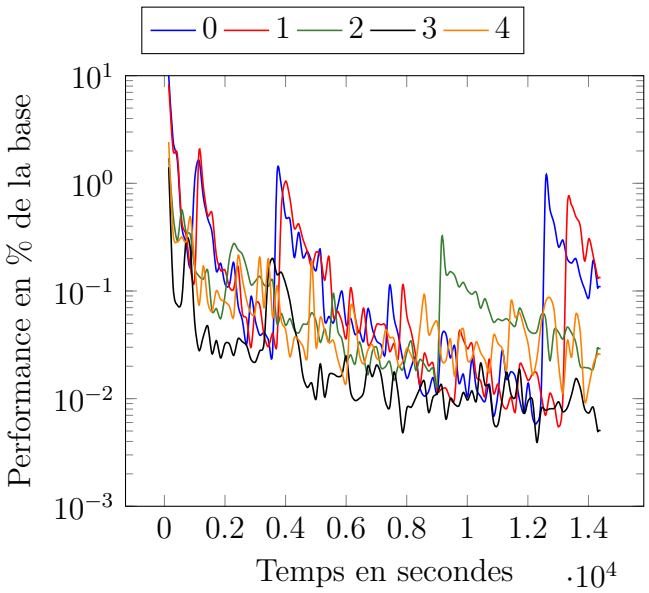


FIGURE 5.9 – Attaque de JtR-Markov sur Myspace avec ZXCVBN

5.5.2.1 Mode Markov de John The Ripper

Les figures 5.8, 5.9 et 5.10 montrent que JtR-Markov, sur toutes les bases de données, est totalement stoppé quand les scores requis sont 3 ou 4, tandis qu'il devient peu performant quand le score est 2. Ce n'est pas uniquement lié au score en lui-même, mais également au fait que John The Ripper est très rapide pour générer des candidats, ce qui permet à bcrypt d'être une très bonne contre-mesure. Cela explique également les lignes droites sur les graphiques : puisque John The Ripper est très rapide, on collecte beaucoup plus de candidats pour une période de

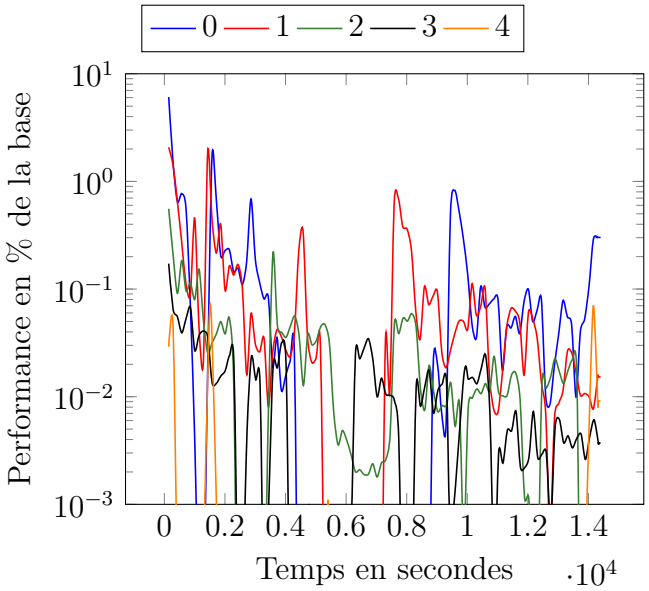


FIGURE 5.10 – Attaque de JtR-Markov sur LinkedIn avec ZXCVBN

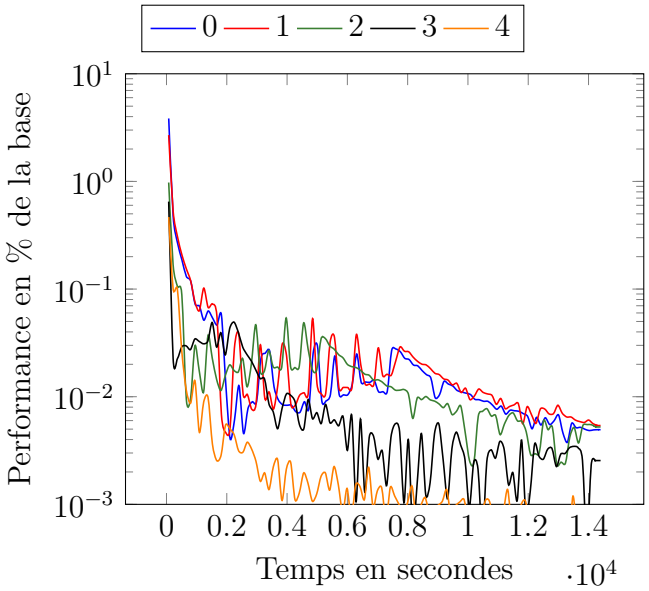


FIGURE 5.11 – Attaque de PCFG sur Rockyou avec ZXCVBN

temps courte, qui prennent du temps à être hachés avec bcrypt. JtR-Markov n'a pas une performance décroissante car comme énoncé dans la section 1.1.5, il parcourt des ensembles de mots tels que la probabilité moyenne des mots de ces ensembles soit décroissante. Puisque JtR-Markov considère des ensembles W_i très grands, la décroissance de performance n'est pas visible sur une petite période de temps.

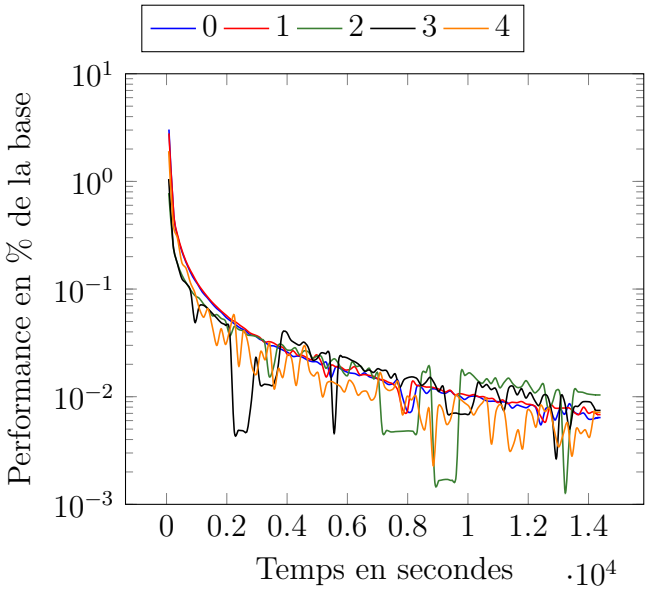


FIGURE 5.12 – Attaque de PCFG sur Myspace avec ZXCVBN

5.5.2.2 PCFG

La figure 5.11 indique que la performance de PCFG est globalement décroissante au fil du temps, ce qui est attendu car les W_i (tels que définis dans la section 1.1.5) qu'il considère sont de taille beaucoup plus petite que ceux de JtR-Markov. On observe que les scores 0, 1 et 2 donnent des performances proches pour PCFG, ce qui signifie qu'ils ne sont pas vraiment efficaces pour protéger les mots de passe de cette attaque. Cependant, après 4000 secondes d'attaque, les scores 3 et 4 font baisser la performance d'environ un ordre de grandeur, devenant ainsi moins de 0.001% de la taille de la base de données.

La figure 5.12 montre que PCFG n'est pas impacté par une politique de score élevé avec Myspace. Le score 4 permet d'avoir une performance de PCFG presque toujours la plus basse, mais ce score ne permet pas de baisser significativement cette performance.

On observe sur la figure 5.13 que les courbes se croisent au fil du temps sur LinkedIn. Au début de l'attaque, les scores 2, 3 et 4 sont plutôt efficaces contre PCFG. Après 3000 secondes et jusqu'à 6000 secondes, ce sont les scores 0 et 1 qui protègent le mieux contre cette attaque. Au delà de 10000 secondes, le score 4 redevient le plus efficace, alors que les scores 2 et 3 sont proches des scores 0 et 1. Les baisses soudaines de performances que l'on peut observer entre 0 et 6000 secondes sont dues à la manière de fonctionner de PCFG : il itère sur les structures indépendamment du nombre de mots de passe trouvés. Dans ces moments, PCFG est bloqué sur une structure peu rentable.

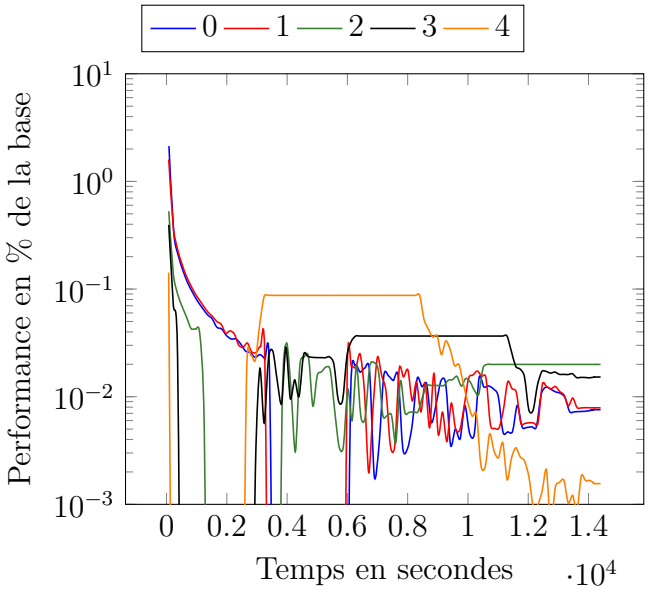


FIGURE 5.13 – Attaque de PCFG sur LinkedIn avec ZXCVBN

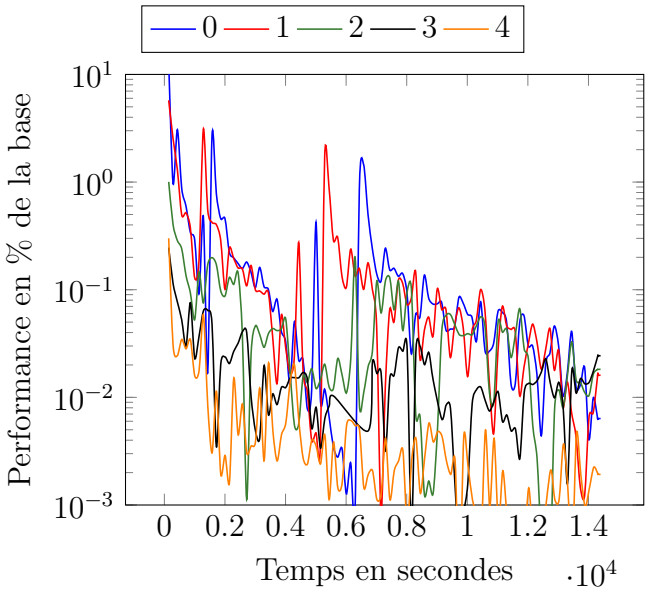


FIGURE 5.14 – Attaque d'OMEN sur Rockyou avec ZXCVBN

5.5.2.3 OMEN

On peut observer que globalement les performances sont liées aux scores sur la figure 5.14. Même si chaque incrémentation du score n'offre pas beaucoup plus de résistance face à OMEN, la différence entre les scores 0 et 4 est considérable (environ 2 ordres de grandeur). De manière surprenante, le score 4 offre moins de protection que le score 3 sur la base Myspace (figure 5.15), même si les performances sont assez proches les unes des autres.

Comme montré sur la figure 5.16, les scores 1 et 2 n'augmentent pas significati-

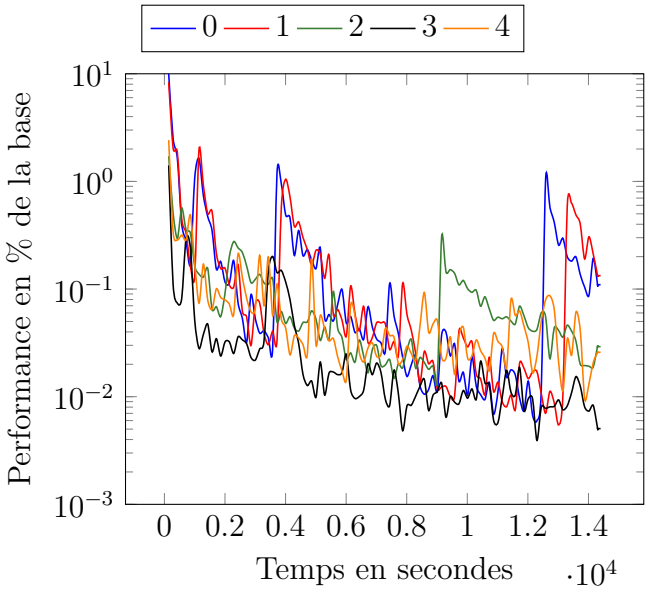


FIGURE 5.15 – Attaque d'OMEN sur Myspace avec ZXCVBN

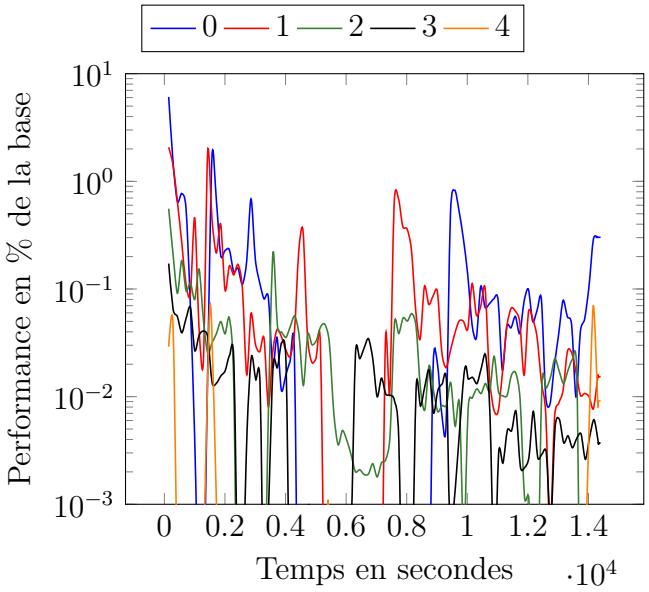


FIGURE 5.16 – Attaque d'OMEN sur LinkedIn avec ZXCVBN

vement la robustesse des mots de passe de LinkedIn face à OMEN. Cependant, les scores 3 et 4 offrent une meilleure protection puisque la performance est presque toujours nulle. Le score 4 offre quasiment une protection complète face à cette attaque. Sur Myspace, ajouter une restriction sur le score de ZXCVBN aux mots de passe en plus de l'usage de bcrypt ne permet pas d'augmenter réellement la robustesse des mots de passe. Cependant, pour Rockyou, les scores 3 et 4 contre PCFG et OMEN permettent de baisser les performances d'environ 2 ordre de grandeur, ce qui signifie que les attaques sont cent fois plus coûteuses. OMEN est totalement inefficace contre LinkedIn avec l'usage du score 4 de ZXCVBN, tandis que PCFG est moins efficace

uniquement après 10000 secondes d'attaque.

5.5.3 Synthèse

Les expériences montrent que même en filtrant les mots de passe avec des métriques de robustesse, les bases de données restent peu résistantes aux attaques des énumérateurs probabilistes. C'est pourquoi nous ne recommandons pas l'utilisation de telles métriques pour protéger les mots de passe des énumérateurs probabilistes.

Même si les expériences indiquent que *ZXCVBN* permet d'obtenir une meilleure résistance des mots de passe face à PCFG et OMEN que les heuristiques basiques, ZXCVBN n'est pas efficace sur tous les ensembles étudiés : Myspace n'est quasiment pas protégée de l'attaque de PCFG et OMEN, même quand le score requis est élevé. Par précaution, ZXCVBN doit être considéré comme inadaptée pour mesurer la robustesse des mots de passe face aux attaques basées sur les énumérateurs PCFG et OMEN.

5.5.4 Compromis sécurité-utilisabilité

Une politique offrant une bonne utilisabilité rejette le moins de mots de passe possible. Ainsi, on peut supposer qu'un filtre qui rejette peu de mots de passe est plus utilisable qu'un filtre qui en rejette beaucoup. ZXCVBN avec un score de 3 ou 4 rejette moins de mots de passe que *complex* et *longcomplex*. De plus, les bases de données sont mieux protégées avec ZXCVBN face aux énumérateurs probabilistes, puisque leur performances sont en moyenne dix fois moins élevées qu'avec les heuristiques basiques. Ceci laisse penser que ZXCVBN sera mieux accepté par les utilisateurs finaux. Comme il n'est pas suffisamment efficace sur toutes les bases de données mais qu'il n'y a pas d'alternative viable pour le moment, nous ne le suggérons qu'en dernier recours avec un score de 3 ou 4 .

5.6 Conclusion

Dans ce chapitre, nous avons montré que les bases de données ne deviennent pas suffisamment résistantes aux attaques des énumérateurs probabilistes, même lors qu'on y retire les mots de passe les plus faibles, en étudiant l'évolution des performances locales des énumérateurs face aux bases de données filtrées. En effet, *complex* et *longcomplex*, qui pourtant sont les heuristiques qui offrent le meilleur compromis sécurité/utilisabilité face aux attaques plus naïves telles que la force brute, ne réduisent pas suffisamment les performances des énumérateurs probabilistes sur les bases de données étudiées. De plus, ZXCVBN, présenté comme une mesure de la robustesse avancée, ne protège pas efficacement les bases de données. Néanmoins, si il est paramétré avec un score de 3 ou 4, alors cela permet d'obtenir une meilleure sécurité et utilisabilité que *complex* et *longcomplex*. C'est pourquoi, ZXCVBN ne doit être utilisé qu'en dernier recours pour protéger les bases de données jusqu'à ce que des métriques plus adaptées soient conçues.

De plus, les bases de données vont de plus en plus être protégées avec des fonctions de hachage lentes, ce qui laisse penser que les énumérateurs probabilistes tels que PCFG et OMEN vont également devenir plus populaires pour casser des mots de passe. Ainsi, il y a besoin de concevoir de nouvelles métriques de robustesse qui soient efficaces contre ces attaques. Les métriques multimodales [15], qui agrègent plusieurs métriques plus élémentaires, sont un pas dans la bonne direction pour renforcer les bases de données face aux attaques novatrices basées sur les énumérateurs probabilistes.

Même s'il semble possible de concevoir des nouvelles politiques de sécurité qui soient suffisamment efficaces pour protéger d'un énumérateur donné, le véritable challenge serait qu'elles soient efficaces contre tous les énumérateurs, tout en garantissant qu'elles soient acceptées par les utilisateurs. On pourrait par exemple naïvement proposer une politique de sécurité $\mathcal{P}_E$ efficace pour chaque énumérateur E. Ces politiques rejetteraient les mots de passe dont la robustesse est inférieure à un seuil. Une politique naturelle $\mathcal{P}$ et efficace contre tous les énumérateurs est celle qui fait passer le mot de passe par chaque politique $\mathcal{P}_E$ et n'accepte que les mots dont la robustesse est supérieure à tous les seuils. Mais $\mathcal{P}$ amène plusieurs problèmes. Premièrement, elle risque de rejeter beaucoup de mots, ce qui frustrera sûrement les utilisateurs. Deuxièmement, comment indiquer à l'utilisateur la raison du rejet ? Et comment lui indiquer les manières de renforcer son mot de passe ?

Chapitre 6

Analyse du comportement des énumérateurs probabilistes

Dans ce chapitre, on analyse le comportement des énumérateurs probabilistes en étudiant l'évolution des structures de données qu'ils utilisent. L'objectif est de comprendre et isoler les paramètres permettant de freiner les performances de ces énumérateurs. Grâce à ce travail, il devient possible de faire en sorte que les mots de passe d'une base de données soient dispersés dans l'espace de recherche de sorte à freiner les performances des énumérateurs.

Sommaire

6.1 Introduction

Dans les deux chapitres précédents, nous avons montré que les énumérateurs probabilistes étaient très efficaces pour attaquer les bases de données de mots de passe. Dans l'optique de proposer de nouvelles manières d'évaluer la robustesse des mots de passe, il est nécessaire de bien comprendre comment fonctionnent ces énumérateurs afin de déterminer les différents moyens de rendre plus coûteuses leurs exécutions. Dans ce chapitre, on étudie deux énumérateurs probabilistes de l'état de l'art : l'implantation de la grammaire de Weir et OMEN. Pour chacun de ces énumérateurs, on présente tout d'abord les algorithmes qui le composent,

tout en identifiant les opérations les plus coûteuses que l'on devra observer durant les analyses, en présentant dans un premier temps l'algorithme d'entraînement du modèle probabiliste et ensuite l'algorithme de génération des candidats. Ensuite, on expose les choix des structures de données pour implanter les algorithmes. Les choix effectués sont selon nous ceux qui permettent au programme d'être le plus rapide possible et le moins coûteux en mémoire. Les expériences sont alors présentées, où les mesures concernant les structures de données sont présentées.

6.2 PCFG de Weir

On rappelle qu'une PCFG est définie comme un quintuplet de 4 ensembles finis et d'un élément, $\langle \Sigma, \mathcal{N}, \mathcal{S}, \mathcal{R}, \mathcal{P} \rangle$, où Σ est un ensemble de symboles terminaux, $\mathcal{N}$ un ensemble de non-terminaux, $\mathcal{S}$ le symbole spécial "début" de $\mathcal{N}$, $\mathcal{R}$ un ensemble de règles de production de la forme $\mathcal{A} \to \mathcal{B}$ avec $\mathcal{A} \in \mathcal{N}$ et $\mathcal{B} \in (\Sigma \cup \mathcal{N})^*$, et $\mathcal{P}$ est l'ensemble des probabilités associées aux règles.

Weir définit $Q = (\mathcal{N} \backslash \mathcal{S})$ [54] comme l'ensemble des symboles non-terminaux sans le symbole "début". Les règles de production de la grammaire sont soit de la forme

$$\mathcal{S} \to \mathcal{Q}, \mathcal{Q} \in Q^*$$

soit de la forme

$$B \to T, B \in Q, T \in \Sigma.$$

Les éléments de Q sont appelés bases simples, et les éléments de Q^* sont appelés bases composées.

Soit w un mot quelconque. On note $w = T_1 ... T_k$ le découpage de w en k terminaux, c'est-à-dire que chaque T_i n'est composé que de lettres provenant d'une seule classe de caractères. On note $\mathcal{Q}_w$ la base composée de w, et $B(T_i)$ la base simple du terminal T_i.

Exemple : Soit $w =$"ordinateur12 !". On a $T_1 =$"ordinateur", $T_2 =$"12", $T_3 =$" !". La base composée $\mathcal{Q}_w$ de ce mot est $\mathcal{Q}_w = L_{10} D_2 S_1$ car il est composé de 10 minuscules suivies de 2 chiffres et terminant par un caractère spécial. Les bases simples de w sont $B(T_1) = L_{10}, B(T_2) = D_2, B(T_3) = S_1$. De ce mot découlent 4 règles :

- $\mathcal{S} \to L_{10} D_2 S_1$
- $L_{10} \to ordinateur$
- $D_2 \to 12$
- $S_1 \to !$

Définition des probabilités : dans une grammaire probabiliste, on définit une distribution de probabilité $p_\mathcal{A}$ pour chaque $\mathcal{A}$ intervenant dans une règle de la forme $\mathcal{A} \to \mathcal{B}$. On définit $p_\mathcal{A}(\mathcal{B})$ la probabilité d'appliquer la règle $\mathcal{A} \to \mathcal{B}$ à partir de $\mathcal{A}$, pour chaque $\mathcal{B}$ tel que $\mathcal{A} \to \mathcal{B}$ est une règle.

Dans le cas où $\mathcal{A} = \mathcal{S}$, $p_\mathcal{S}$ est bien un distribution de probabilité associée à $\mathcal{S}$ car $\sum p_\mathcal{S}(\mathcal{B}) = 1$ pour tout $\mathcal{B}$ intervenant dans une règle de la forme $\mathcal{S} \to \mathcal{B}$.

Soit $B = B(T_j)$ la base simple d'un terminal T_j. On définit $p_B(T_j)$ comme étant le rapport entre le nombre d'occurrences de T_j divisé par le nombre d'occurrences de B dans la base de données d'apprentissage.

6.2.1 Algorithme d'apprentissage

Cet algorithme va parcourir la base de données d'entraînement $\mathcal{D}$ et calculer les statistiques sur toutes les règles rencontrées, c'est-à-dire la fréquence d'apparition de chaque règle. On suppose que les probabilités $p_{\mathcal{A}}$ correspondent aux fréquences observées dans $\mathcal{D}$. Cet algorithme calcule donc les probabilités p_S et p_B pour chaque base simple B. Pour chaque mot $w \in \mathcal{D}$, l'algorithme calcule la base composée $\mathcal{Q}_w$ associée à w. $\mathcal{Q}_w$ est calculée est parcourant les caractères un à un. Une fois $\mathcal{Q}_w$ calculée, on incrémente les compteurs $acc[\mathcal{S}]$ (le nombre total de bases composées) et $acc[\mathcal{Q}_w]$ associés. Ensuite, pour le même mot w, l'algorithme découpe $\mathcal{Q}_w$ en k bases simples, telle que $\mathcal{Q}_w = B_1...B_k = B(T_1)...B(T_k)$. Pour chaque terminal T_i de w, on calcul $B = B(T_i)$ et on incrémente les compteurs $acc[B]$ et $acc[T_i]$. L'algorithme ici ne fait pas de distinction entre les minuscules et les majuscules.

Une fois $\mathcal{D}$ totalement parcourue, l'algorithme calcule les distributions de probabilités. Pour chaque base composée $\mathcal{Q}_i$, on a

$$p_S(\mathcal{Q}_i) = \frac{acc[\mathcal{Q}_i]}{acc[\mathcal{S}]}$$

$p_S(\mathcal{Q}_i)$ est le ratio entre le nombre d'occurrences de la base composée $\mathcal{Q}_i$ et le nombre total de bases composées.

Pour chaque terminal T_i, on note $B = B(T_i)$ sa base simple. On a :

$$p_B(T_i) = \frac{acc[T_i]}{acc[B]}$$

$p_B(T_i)$ est le ratio entre le nombre d'occurrences du terminal T_i et le nombre total de bases simples B. C'est lors du calcul de ces deux probabilités que l'on suppose que la probabilité reflète la fréquence observée dans $\mathcal{D}$ des règles.

Ensuite, puisque la production d'un mot $w = T_1...T_k$ se fait de manière unique, et que l'ensemble $\mathcal{W}$ de tous les mots produits par la grammaire est un produit cartésien, on peut calculer la probabilité $p(w)$ en effectuant le produit des distributions de probabilités utilisées pour décomposer w en terminaux, tel que

$$p(w) = p_S(\mathcal{Q}) \prod_{i=1}^{k} p_{B_i}(T_i). \tag{6.1}$$

Pour gagner en espace mémoire et en vitesse d'exécution, l'algorithme peut arrondir les probabilités, par un procédé appelé "smoothing" dans la publication de Weir. Notons, cela ne permet plus de garantir la génération des candidats par ordre de probabilité décroissante.

6.2.2 Algorithme de génération

Cet algorithme va faire en sorte de produire les mots de la grammaire entraînée dans un ordre décroissant de probabilité, à moins que les probabilités n'aient été arrondies. Pour cela, l'algorithme se repose sur une file de priorité, où les éléments sont

ordonnés par probabilité décroissante. Chaque élément de la file est un quadruplet $\langle \mathcal{Q}, w, p(w), piv \rangle$ où $\mathcal{Q}$ est une base composée, w le prochain candidat de $\mathcal{Q}$ à être généré, $p(w)$ la probabilité de w, et piv une valeur pivot déterminant la prochaine base simple sur laquelle appliquer une règle.

Initialisation. L'algorithme initialise la file de priorité F de cette manière.

Pour chaque base composée $\mathcal{Q}$, on calcule le candidat le plus probable ayant pour base composée $\mathcal{Q}$. Pour cela, on applique la règle r_i la plus probable de la forme $B \to T$, pour chaque base simple B de $\mathcal{Q}$.

On calcule $p(w)$ en fonction de la probabilité de $\mathcal{Q}$ et celles des règles r :

$$p(w) = p_{\mathcal{S}}(\mathcal{Q}).\prod_{i=1}^{k} p_B(T_i)$$

telle que définie dans l'équation 6.1. Le pivot est initialisé à 0 indiquant que la prochaine règle à appliquer concerne la première base simple. On insère dans F le quadruplet $\langle \mathcal{Q}, w, p(w), piv \rangle$.

Génération. L'algorithme de génération peut prendre un paramètre α en entrée, correspondant au seuil de probabilité au delà duquel les éléments ne sont pas insérés dans la file de priorité.

On dépile $\langle \mathcal{Q}, w, p(w), piv \rangle$ de F.

On affiche w, puis on calcule les prochains candidats en continuant l'application des règles de la manière suivante : pour chaque base simple B telle que sa position $pos \geq piv$ dans $\mathcal{Q}$, on accède à la règle r de la forme $B \to T_i$ appliquée juste avant de produire w en utilisant un pointeur, et on récupère $r' = B \to T_j$ la règle qui suit r dans un ordre de probabilité décroissante.

On applique r' pour produire w'. On calcule

$$p(w') = p(w).\frac{p_B(T_j)}{p_B(T_i)}. \tag{6.2}$$

On insère dans F le quadruplet $\langle \mathcal{Q}, w', p(w'), pos \rangle$ si $p(w') > \alpha$ en conservant l'ordre des éléments (probabilité décroissante).

Utilisation de pointeurs. Afin d'être efficace dans la génération des candidats, il est plus intéressant d'utiliser des pointeurs pour indiquer la progression de l'algorithme dans les règles à appliquer, au lieu de stocker la règle elle même dans le quadruplet. Dans un premier temps, les règles sont stockées dans un dictionnaire $Dict$ où une clé est une base simple B et où une valeur est la liste de terminaux T_i intervenants dans les règles du type $B \to T_i$. Ces listes sont triées par ordre décroissant de probabilité. Par exemple, on accède à la liste des terminaux de la base simple L_8 en utilisant $Dict[L_8]$. Ainsi, $Dict[L_8][0]$ pointe le terminal le plus probable de L_8.

Ensuite, pour chaque base simple B d'une base composée $\mathcal{Q}$, on mémorise dans un tableau la position m, appelé pointeur, de la dernière règle de la forme $B \to T$ appliquée concernant B. Ainsi, à chaque fois que l'on souhaite appliquer la règle suivante, il suffit d'incrémenter le pointeur. C'est seulement à l'affichage du candidat qu'il est nécessaire de récupérer les terminaux correspondants aux règles désignées par les pointeurs. Par exemple, si l'on considère $\mathcal{Q} = L_8 D_2 L_2 D_2$, le tableau des

positions des règles concernant L_8, D_2, L_2, D_2 est initialisé à $[0, 0, 0, 0]$, on sait donc que c'est la première règle de chaque base simple qui est appliquée. Lorsque l'on souhaite appliquer la règle suivante pour L_8, il suffit d'incrémenter le premier nombre du tableau, en s'assurant qu'on ne dépasse pas le nombre de terminaux de L_8.

Préterminal : Weir introduit une notion dans son algorithme de génération de candidat, appelée préterminal.

Un préterminal est un mot où l'on ne va pas appliquer les règles concernant un type de base simple particulier. Par exemple, le mot $L_{10}12!$ est un préterminal car la base simple L_{10} n'a jamais été remplacée. Ainsi, au lieu de générer des mots candidats où au moins une règle est appliquée par base simple, l'algorithme de Weir va générer des préterminaux où il reste des bases simples de la forme L_i. Ceci permet de terminer l'application des règles pour les bases simples restantes de manière massivement parallèle sur des cartes graphiques par exemple, et ainsi générer plusieurs candidats à la suite pour un même élément de la file.

Les règles restantes, de la forme $L_i \to T_j$ sont soit calculées avec la grammaire, soit en utilisant une ressource supplémentaire tel un dictionnaire. L'inconvénient est que l'application des règles de manière parallèle ne permet plus de générer les candidats par ordre décroissant de probabilités, car toutes les règles sont appliquées en même temps. L'algorithme prend donc chaque préterminal par ordre de probabilité décroissante et applique l'ensemble des règles restantes, jusqu'à un certain seuil de probabilité.

Dans notre cas, nous n'utilisons pas ces préterminaux et générons directement les terminaux, car l'aspect parallèle ne nous intéresse pas pour étudier la file de priorité.

6.2.3 Structure de données

Dans cet algorithme de génération de candidats, la structure de données principalement utilisée est la file de priorité. À chaque fois que l'on génère un candidat, on effectue ces opérations :

1. défilage d'un élément ;
2. application de l règles ;
3. calcul de l probabilités, deux produits de nombres flottants (équation 6.2) ;
4. enfilage de l éléments.

où l est le nombre de bases simples dont la position est supérieure au pivot et pour lesquelles il reste des règles à appliquer. Dans le cas où il n'y a plus de règle à appliquer pour la base simple concernée, l'algorithme passe à la base simple suivante sans insérer d'élément. L'opération 1 se fait en temps constant, car la file est triée. La deuxième étape dépend de l, et la troisième se résume à $2l$ produits. La dernière étape dépend également de l, mais aussi de la taille s de la file, car il faut chercher l'endroit où placer les éléments insérés. Une insertion se fait donc en $log(s)$ comparaisons de deux éléments de F (recherche par dichotomie). À chaque candidat généré, on insert donc l éléments, ce qui détermine l'agrandissement de la file. Le coût de l'énumération est donc essentiellement dans l'enfilage des éléments, l'application des règles et la mise à jour des probabilités.

L'étape de mise à jour de la probabilité du prochain candidat s'effectue en deux opérations (équation 6.2) : une division de l'ancienne probabilité par la probabilité de la dernière règle appliquée puis un produit avec la nouvelle règle appliquée. Retrouver ces deux règles se fait en deux accès directs en utilisant les pointeurs. On suppose que ces deux opérations sont négligeables par rapport à l'opération 4.

C'est donc essentiellement la taille de la file de priorité qui détermine le coût de génération de chaque candidat.

6.2.4 Analyse de l'évolution de la file de priorité

Dans cette partie, on étudie la taille de la file de priorité de l'algorithme de Weir durant l'énumération et en fonction de différents paramètres telle que la base de données d'apprentissage. Pour cela, on lance l'algorithme d'énumération et on affiche après $N = 10^6$ candidats la taille de la file de priorité, car c'est moins coûteux que de l'afficher après chaque candidat. On affiche également le nombre de mots trouvés avec ces N candidats, afin de garder un œil sur la difficulté à trouver des mots de passe durant l'énumération. Nous avons utilisé notre propre implantation de l'algorithme de Weir, en C++, afin de maîtriser totalement les opérations effectuées par le programme et également d'être bien plus rapide que l'implantation en Python originale. La taille des files sont différentes de l'implantation originale effectuée par Weir car celui-ci introduit des compromis afin de réduire l'espace mémoire utilisé au détriment de proposer des candidats dans un ordre décroissant de probabilité, comme le fait de terminer l'application des règles sur des cartes graphiques.

6.2.4.1 Choix des paramètres

Nous avons choisi d'effectuer la mesure de la taille de la file jusqu'à 10^9 candidats, car soit au delà la file continue de grandir de la même manière, ou bien dans le cas où α n'est pas trop petit, l'algorithme termine avant. Pour rappel, α est le seuil de probabilité en dessous duquel les éléments ne sont pas insérés dans la file de priorité.

Bases de données : trois bases de données sont utilisées dans ces expériences : Rockyou, LinkedIn et Myspace. Nous avons partitionné chaque base de données de manière aléatoire, une partie pour l'apprentissage et une partie pour l'évaluation et la mesure du nombre de mots trouvés.

Paramètre α : nous avons analysé deux valeurs de α : 0 (aucune restriction sur $p(w)$), et 10^{-10}. Les autres valeurs essayées n'ont pas permis d'observer de différence majeure avec l'une des deux précédentes.

PathWell : nous avons également testé le filtrage des bases avec PathWell [31] pour empêcher l'utilisation des bases composées trop populaires. Pour cela, nous avons fixé un seuil de popularité au delà duquel une base composée n'est plus autorisée. Par exemple si $seuil = 100$, alors chaque base composée peut être utilisée uniquement 100 fois. Nous avons réalisé l'expérience pour un seuil de 10^4 et 10^2, pour lesquels nous fixons $\alpha = 0$. D'autres valeurs de seuil ont également été étudiées, mais n'ont pas permis d'observer de différence majeure avec celles présentées.

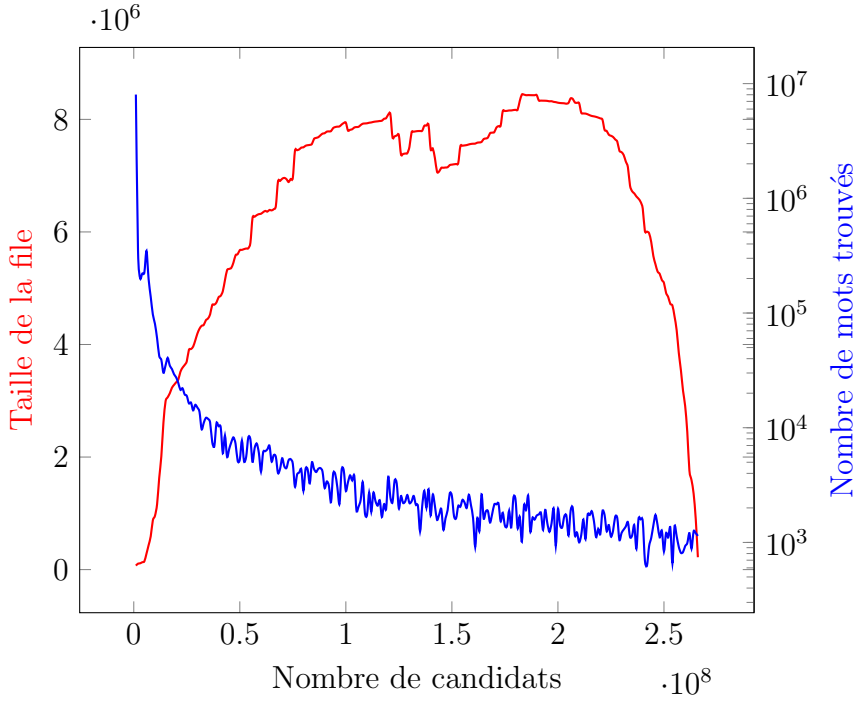


FIGURE 6.1 – Rockyou avec $\alpha = 10^{-10}$

6.2.4.2 Résultats en faisant varier α

Sur Rockyou avec $\alpha = 10^{-10}$ (Figure 6.1), PCFG termine après avoir généré un peu plus de 2.5×10^8 candidats. La file de priorité grandit jusqu'à environ 2×10^8 candidats et ensuite la file se vide. Au plus haut, la taille de la file atteint 8×10^6 éléments. Le nombre de mots trouvés décroît de moins en moins vite, jusqu'à atteindre environ 10^3 mots trouvés tous les 10^6 candidats générés. Avec $\alpha = 0$ (Figure 6.2), PCFG atteint la limite des 10^9 candidats que l'on s'est fixée. Au début de l'énumération et jusqu'à environ 0.2×10^9 candidats, la file grandit de manière quasi-linéaire, avec un accroissement de la taille plus important au tout début. Cependant, on remarque que la file commence à grandir de plus en plus à partir de 0.4×10^9 candidats. Le nombre de mots trouvés au cours de l'énumération décroît de la même manière qu'avec $\alpha = 10^{-10}$.

En attaquant LinkedIn avec $\alpha = 10^{-10}$ (Figure 6.3), PCFG génère un peu plus de 3.5×10^8 candidats. Le pic de la taille de la file se situe à 3×10^8 candidats et vaut environ 1.8×10^7 éléments. Cependant, la croissance de la file est différente de celle contre Rockyou et $\alpha = 10^{-10}$. Même si elle évolue en deux temps comme avec Rockyou, ici elle augmente vite jusqu'à 10^8 puis moins vite jusqu'à 3×10^8, alors qu'avec Rockyou, la file augmente vite jusqu'à 10^8 mais oscille ensuite entre 7×10^6 et 8×10^6 éléments jusqu'à 2.5×10^8 candidats. La courbe du nombre de mots trouvés montre une baisse drastique au tout début de l'énumération, probablement dû au fait qu'une seule structure a été utilisée et qu'elle n'a pas été très fructueuse.

Avec $\alpha = 0$ (Figure 6.4), la taille de la file augmente de moins en moins au cours de l'énumération. Contrairement à Rockyou, la croissance ne semble pas augmenter

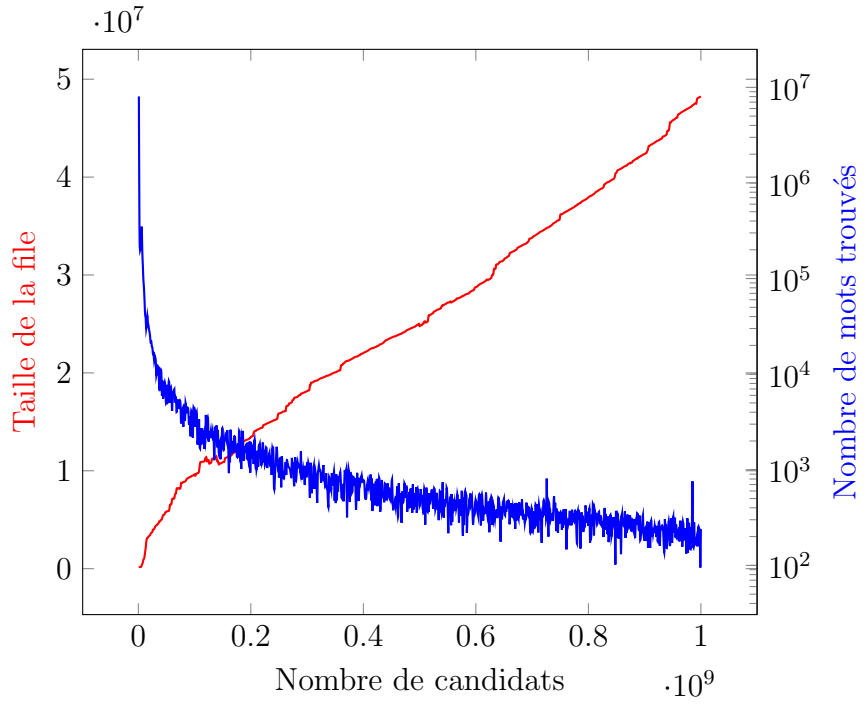


FIGURE 6.2 – Rockyou avec $\alpha = 0$

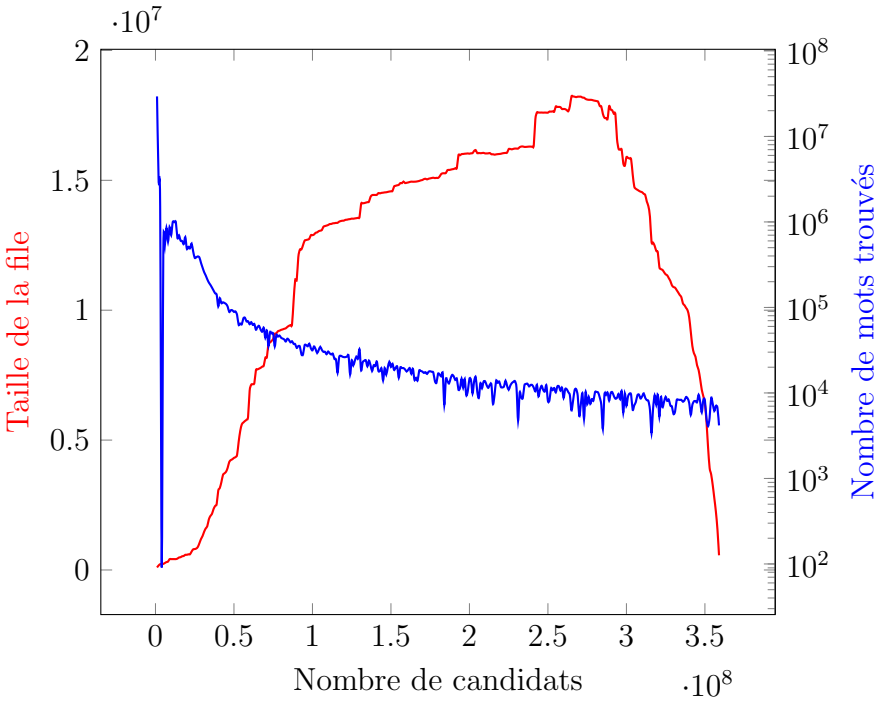


FIGURE 6.3 – LinkedIn avec $\alpha = 10^{-10}$

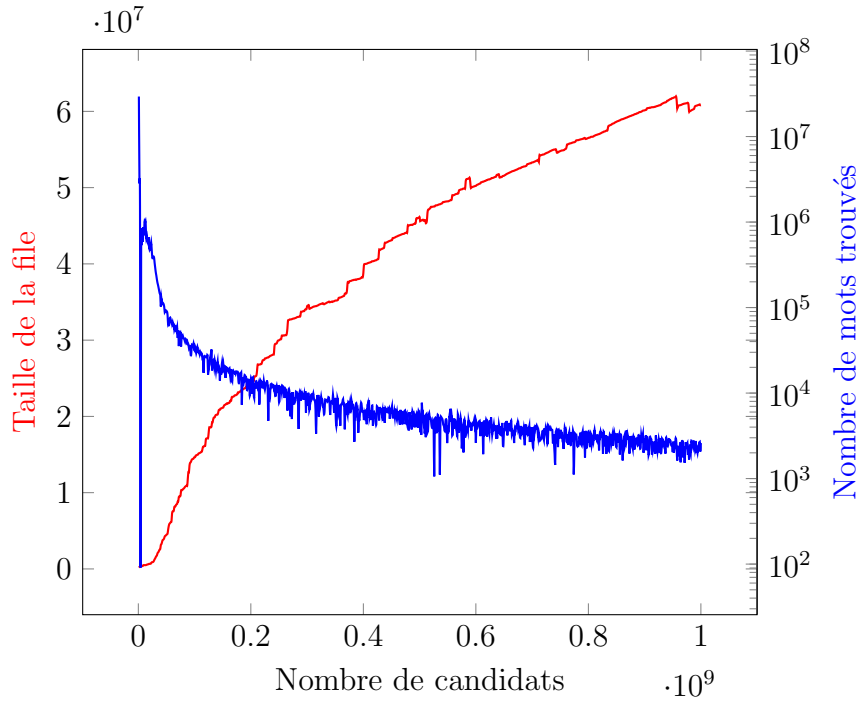


FIGURE 6.4 – LinkedIn avec $\alpha = 0$

après 0.4×10^9 candidats. Cela signifie que la difficulté à générer des candidats n'augmente pas linéairement en le nombre de candidats générés.

Sur Myspace avec $\alpha = 10^{-10}$ (Figure 6.6), la taille de la file de priorité décrit sensiblement la même cloche que pour Rockyou. Cependant, avec $\alpha = 0$ (Figure 6.5), la file grandit quasi-linéairement au cours de l'énumération, avec semble-t-il trois phases : entre 0 et 0.075×10^9 candidats, où la file grandit assez vite, entre 0.075×10^9 et 0.4×10^9 où la file grandit un peu moins vite, et après où la file grandit encore moins vite. On peut résumer que sur Rockyou et Myspace, après 0.4×10^9 candidats, la file grandit de manière linéaire.

On rappelle que le coût de l'insertion d'un élément dans la file de priorité s'effectue en un nombre logarithmique en la taille de la file de comparaisons d'éléments. Ceci signifie que si la taille de la file grandit de manière linéaire, alors le coût d'insertion d'un élément grandit de manière logarithmique. Doubler la taille de la file ne rend donc pas l'insertion deux fois plus coûteuse mais $log(2)$ fois plus coûteuse.

Dans le cas où l'on souhaite augmenter le coût de l'attaque utilisant PCFG, les objectifs sont donc les suivants :

(1) Croissance exponentielle : si la taille de la file croît de manière exponentielle, alors le coût d'insertion des éléments croît de manière linéaire. Dans ce cas, générer deux fois plus de candidats est donc deux fois plus coûteux en insertions.

(2) Pic de taille au plus tôt : Plus la valeur maximale de la taille de la file est atteinte rapidement dans l'énumération, plus le coût d'insertion augmente rapidement. Par exemple, si au lieu d'atteindre 5×10^7 éléments en 10^9 candidats, cette valeur est atteinte en 10^8 candidats, alors les insertions deviennent $log(10)$ fois plus coûteuses à partir de ce moment.

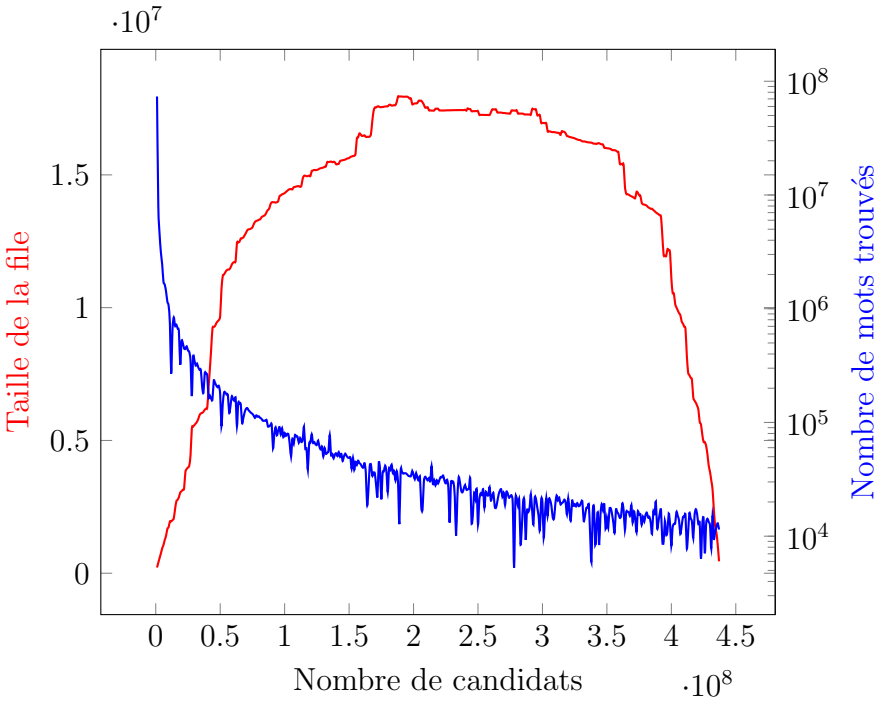


FIGURE 6.5 – Myspace avec $\alpha = 10^{-10}$

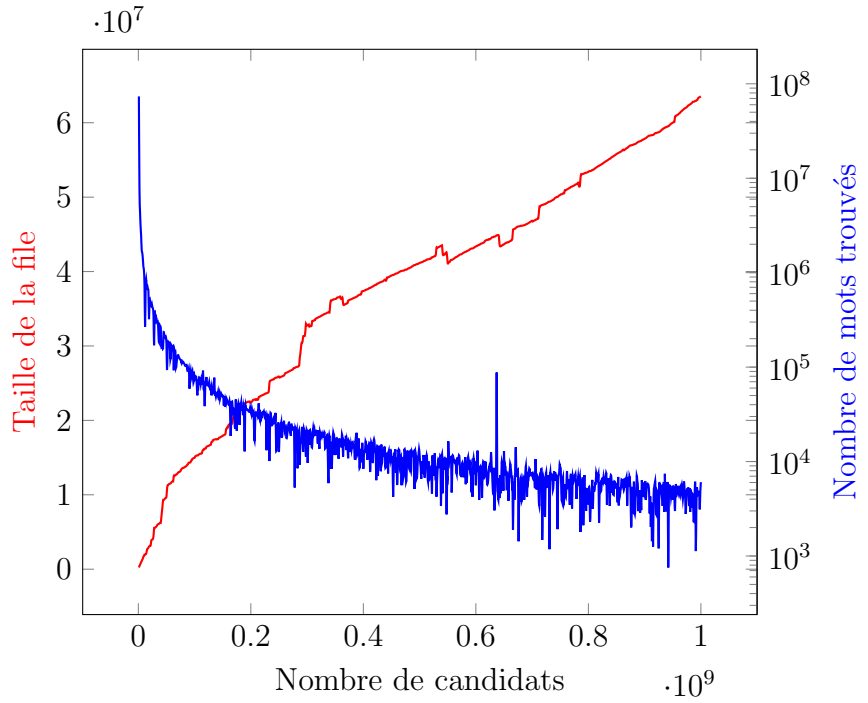


FIGURE 6.6 – Myspace avec $\alpha = 0$

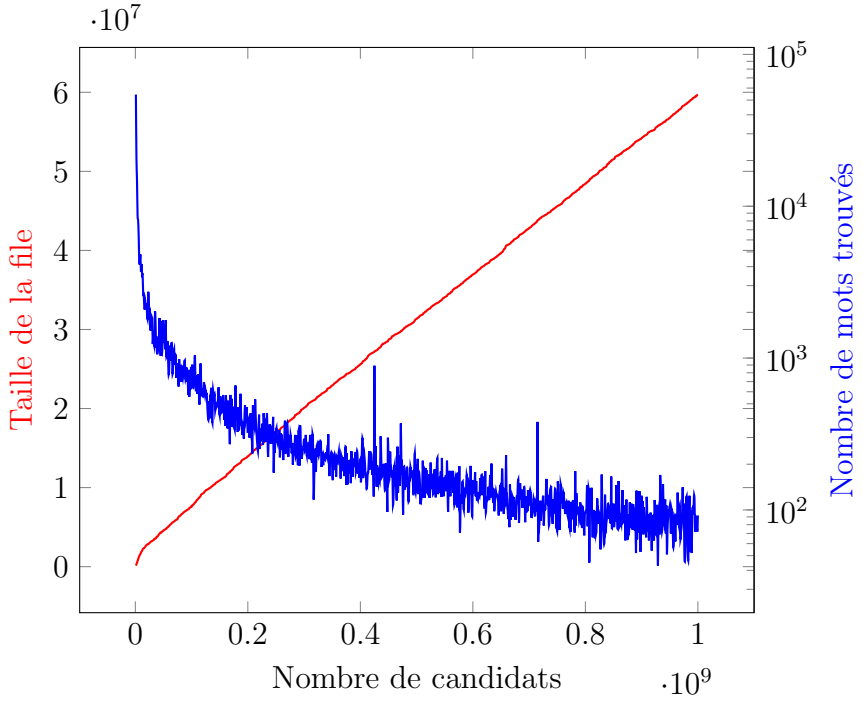


FIGURE 6.7 – Rockyou avec $seuil = 10^4$

(3) Décroissance tardive : lorsque α est paramétrée de sorte à obtenir un nombre raisonnable de candidats générés (de l'ordre de 10^8 par exemple), on veut que le moment où la file commence à se vider se situe le plus tard possible.

La courbe idéale est celle qui est au plus haut durant toute l'énumération et chute très rapidement à la fin. Cette courbe est possible uniquement quand toutes les probabilités des structures et des règles sont égales. Cependant, atteindre ce cas est très difficile, car il faudrait n'autoriser qu'une seule utilisation de chaque base composée et de chaque terminal, ce qui ne semble pas faisable vis-à-vis de l'utilisabilité. Une solution envisageable serait d'utiliser PathWell [31] afin d'interdire l'utilisation d'une base composée trop populaire.

6.2.4.3 Résultats en filtrant les bases avec PathWell

Si l'on filtre la base Rockyou avec PathWell et un seuil de 10^4 (Figure 6.7), on remarque que la taille de la file de priorité grandit de manière linéaire, jusqu'à 6×10^7 en 10^9 candidats. C'est autant que sans PathWell, mais la croissance est différente. La nombre de mots trouvés en revanche est 10 fois moins important.

Avec un seuil à 10^2 (Figure 6.8), la taille de la file est plus de 2 fois plus grande qu'avec un seuil à 10^4 au bout de 10^9 candidats, avec une croissance de file toujours presque linéaire, ce qui veut également dire qu'à tout moment, la file est 2 fois plus grosse avec 10^2 qu'avec 10^4. On peut donc dire que le coût de l'attaque est multiplié par $log(2)$. Le nombre de mots trouvés a encore diminué de plus d'un ordre de grandeur par rapport au seuil de 10^4. L'attaque est donc 10 fois moins fructueuse et $log(2)$ fois plus coûteuse.

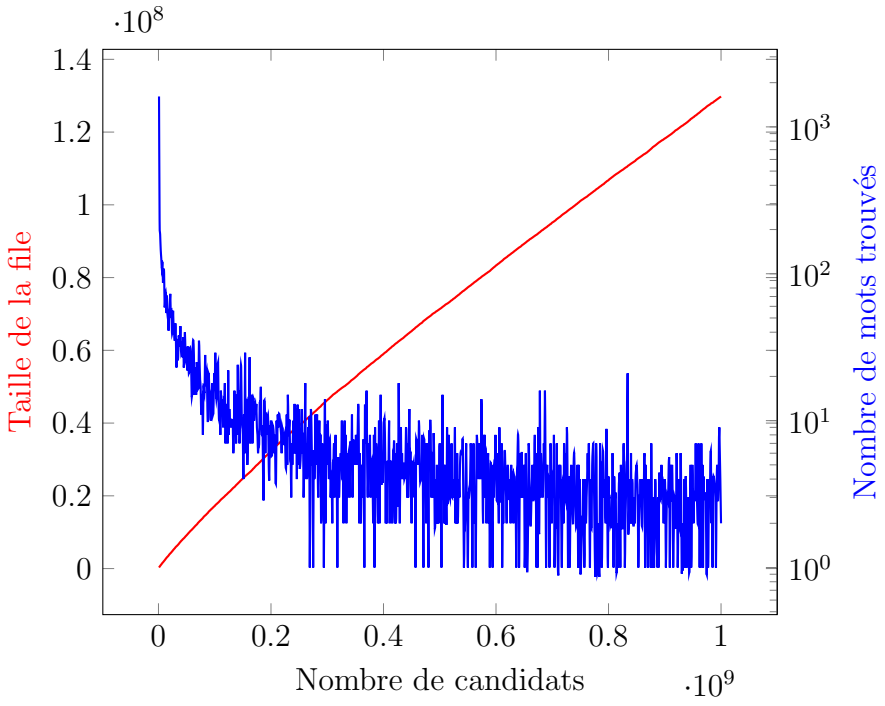


FIGURE 6.8 – Rockyou avec *seuil* $= 10^2$

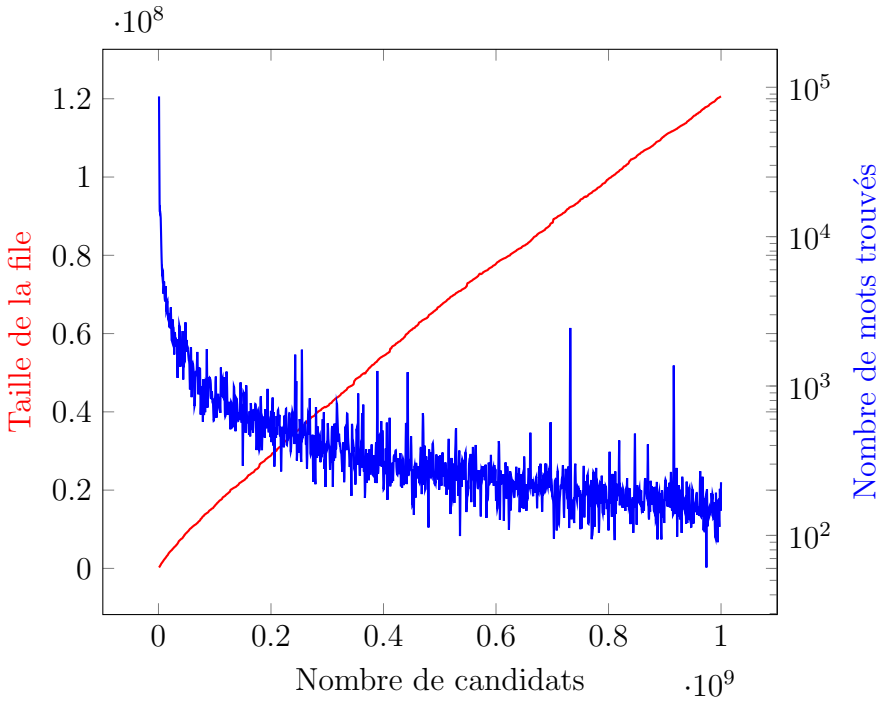


FIGURE 6.9 – LinkedIn avec *seuil* $= 10^4$

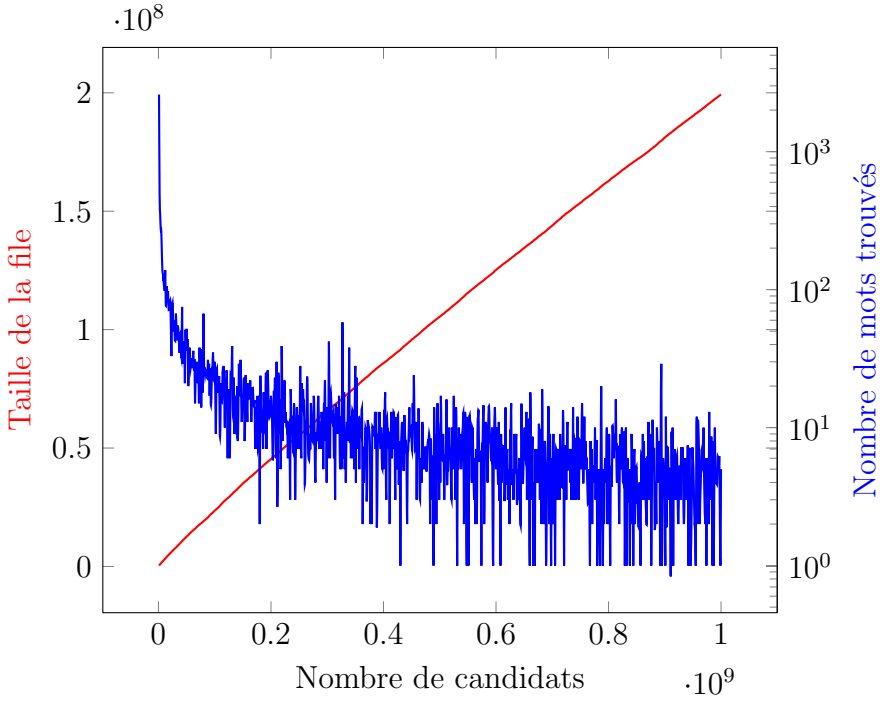


FIGURE 6.10 – LinkedIn avec $seuil = 10^2$

Contrairement à Rockyou, l'application du filtre avec seuil à 10^4 sur la base LinkedIn fait doubler la taille de la file après 10^9 candidats (Figure 6.9) par rapport à la taille de la file sans filtre. Cela rend également linéaire l'agrandissement de celle ci, ce qui n'était pas le cas sans filtre. Le nombre de mots trouvés diminue d'environ un ordre de grandeur par rapport à l'attaque sur la base non filtrée.

Avec un seuil à 10^2 (Figure 6.10), la taille de la file en attaquant LinkedIn double presque de taille par rapport à 10^4, là où le nombre de mots trouvés diminue encore d'un ordre de grandeur.

Tout comme avec LinkedIn, l'application du filtre PathWell avec un seuil à 10^4 sur Myspace (Figure 6.11) fait presque doubler la taille de la file à 10^9 candidats, et permet également de la faire grandir de manière linéaire.

Avec un seuil à 10^2 (Figure 6.12), la file grandit jusqu'à 1.6×10^8 éléments en 10^9 candidats. Le nombre de mots trouvés diminue aussi d'environ un ordre de grandeur.

Sur ces 3 bases de données, l'utilisation de PathWell permet de faire grandir la file de priorité de PCFG plus vite, jusqu'à presque 4 fois plus vite pour LinkedIn, mais aussi de réduire le nombre de mots trouvés de jusqu'à deux ordres de grandeur. Cependant, cela requiert de paramétrer un seuil suffisamment bas, car au delà de 10^2 les effets de PathWell se réduisent beaucoup. Un effet intéressant cependant est le fait que l'agrandissement de la file devienne linéaire, ce qui rend la taille de la file plus prédictible, car dans le cas des bases non filtrées, la taille de la file n'augmente pas de manière linéaire en le nombre de candidats générés, mais semble avoir une allure de courbe logarithmique : il devient proportionnellement plus facile d'insérer des éléments après 10^9 candidats qu'à 10^8 candidats. Il faut tout de même se rappeler que seules les bases composées sont limitées en nombre d'utilisation, et pas les terminaux.

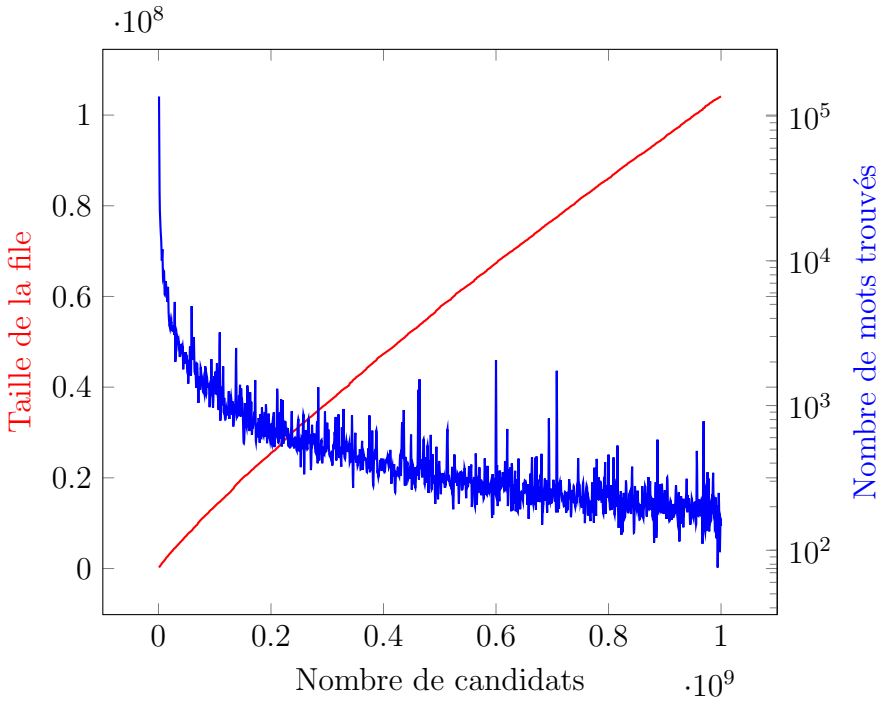


FIGURE 6.11 – Myspace avec *seuil* $= 10^4$

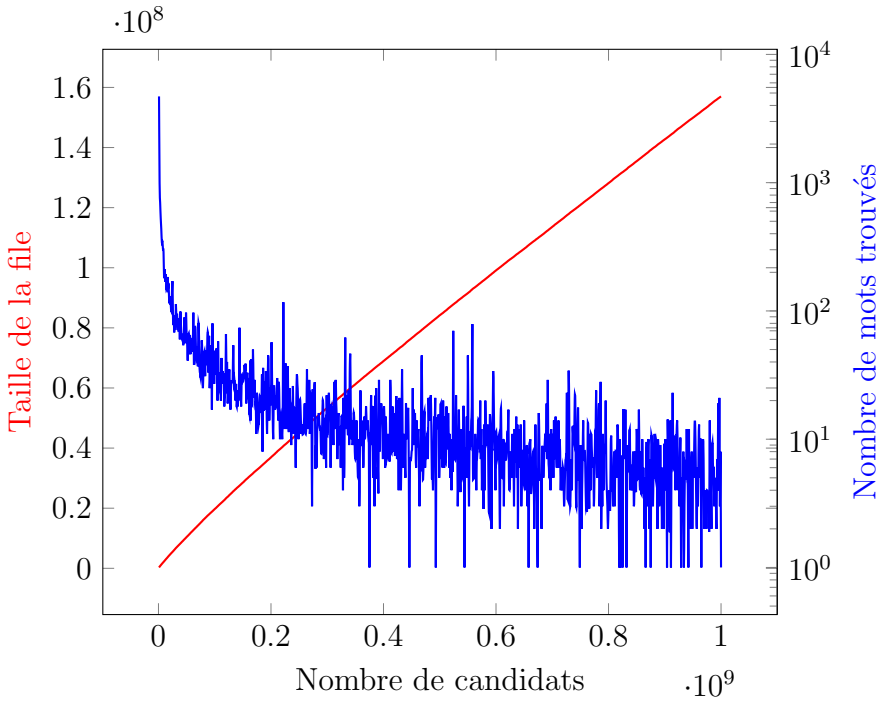


FIGURE 6.12 – Myspace avec *seuil* $= 10^2$

6.3 OMEN

On rappelle qu'OMEN est un algorithme de génération de candidats basé sur un modèle de chaîne de Markov. Ces modèles permettent d'estimer la probabilité d'apparition d'une lettre de l'alphabet en fonction des n lettres qui la précèdent dans le mot. On note $\rho_n(m)$ la probabilité du mot m dans le modèle. Ainsi, on a

$$\rho_n(m) = p(c_1)p(c_2|c_1)...p(c_k|c_{k-n}..c_{k-1}) = \prod_{i=1}^{k} p(c_i|c_{i-n}..c_{i-1})$$

où n est appelé l'ordre de la chaîne de Markov, paramètre déterminant la mémoire du processus. On appelle n-gram un mot de taille n. Ainsi, les n-grams dans un mot de taille l sont au nombre de $l - n + 1$. On note "|" l'opérateur binaire de chevauchement de deux n-gram. La décomposition en n-gram d'un mot w s'écrit $w = w_1|w_2|...|w_{l-n+1}$.

6.3.1 Algorithme d'apprentissage

Cet algorithme va parcourir la base de données d'entraînement $\mathcal{D}$ et calculer les fréquences de tous les n-grams. Pour cela, pour chaque mot w de la base de données, l'algorithme extrait ses $l - n + 1$ n-grams et incrémente le compteur associé. Ensuite, au lieu de calculer la probabilité $p(g)$ de chaque n-gram g, OMEN va discrétiser ces probabilités et les placer dans un nombre fixe H de niveaux. Les niveaux prennent leurs valeurs dans $\{0, 1, ..., H - 1\}$. Chaque niveau η correspond à un intervalle de probabilité tel que $\eta(g) = \lceil -log(c_1.p(g) + c_2) \rfloor$ où c_1 et c_2 sont choisis de telle sorte que le n-gram le plus fréquent soit de niveau 0 et celui le moins fréquent soit de niveau $H - 1$ [1]. Si l'on note p_{max} la probabilité du n-gram le plus probable et p_{min} celle du n-gram le moins probable, alors

$$c_1 = \frac{exp(1 - H) - 1}{p_{min} - p_{max}}$$

et

$$c_2 = 1 - c_1.p_{max}.$$

1. Dans l'article original, les niveaux sont négatifs

Preuve : On veut que $\eta(p_{max}) = 0$ et $\eta(p_{min}) = H - 1$. On résout le système

$$\begin{cases} -log(c_1.p_{max} + c_2) = 0 \\ -log(c_1.p_{min} + c_2) = H - 1 \end{cases}$$

$$\Rightarrow \begin{cases} log(c_1.p_{max} + c_2) = 0 \\ log(c_1.p_{min} + c_2) = 1 - H \end{cases}$$

$$\Rightarrow \begin{cases} c_1.p_{max} + c_2 = 1 \\ c_1.p_{min} + c_2 = exp(1 - H) \end{cases}$$

$$\Rightarrow \begin{cases} c_2 = 1 - c_1.p_{max} \\ c_1.p_{min} + c_2 = exp(1 - H) \end{cases}$$

$$\Rightarrow \begin{cases} c_2 = 1 - c_1.p_{max} \\ c_1.p_{min} + 1 - c_1.p_{max} = exp(1 - H) \end{cases}$$

$$\Rightarrow \begin{cases} c_2 = 1 - c_1.p_{max} \\ 1 - c_1(p_{min} - p_{max}) = exp(1 - H) \end{cases}$$

$$\Rightarrow \begin{cases} c_2 = 1 - c_1.p_{max} \\ c_1 = \frac{exp(1-H)-1}{(p_{min}-p_{max})} \end{cases}$$

Le choix du nombre de niveaux a un impact à la fois sur la précision de l'algorithme et son temps d'exécution. Plus il y a de niveaux, plus l'ordre des mots générés est proche de celui du modèle, mais plus le temps d'exécution est long.

6.3.2 Algorithme d'énumération

L'algorithme d'énumération se résume principalement à une boucle appelant la fonction **enumPwd(η, l)** où l est la taille et η la somme des niveaux des n-grams des mots à générer. Pour cela, **enumPwd** calcule tous les vecteurs $a = (a_1, ..., a_{l-n+1})$ tels que

$$\sum_{i=1}^{l-n+1} a_i = \eta$$

Ensuite, pour chaque vecteur a, il génère tous les mots de la forme

$$w = w_1|w_2|...|w_{l-n+1}$$

où chaque w_i est un n-gram de niveau a_i.

L'algorithme d'énumération est donc une boucle appellant la fonction **enumPwd** en organisant les paramètres η et l de manière à s'adapter au taux de succès au cours de l'énumération.

1. Pour cela, OMEN exécute dans un premier temps **enumPwd(0,l)** pour chaque valeur de l considérée (entre 3 et 20 par défaut) et calcule le taux de succès sp, c'est-à-dire le rapport entre le nombre de mots trouvés dans la base de données ciblée et le nombre de candidats générés. Chaque taux de succès est stocké dans une liste L de taille n (le nombre de tailles différentes), où chaque élément est le triplet (sp, η, l) et où L est triée par sp décroissant.
2. OMEN choisit le triplet le plus probable, $(sp, \eta, l) = L[0]$, exécute

$$enumPwd(\eta + 1, l)$$

et calcule son taux de succès sp'. Il ajoute l'élément $(sp', \eta + 1, l)$ à L en préservant l'ordre et recommence cette étape jusqu'à ce que L soit vide.

6.3.3 Structures de données

Dans un premier temps, OMEN maintient une liste de priorité pour s'adapter à la base de données attaquée. Cette liste est de taille constante, elle dépend uniquement du nombre de valeurs différentes de l qui est un paramètre d'OMEN. De plus, cette liste est relativement courte, car on ne souhaite rarement générer des mots de très grande taille, et ses éléments sont simplement des triplets d'entiers petits.

C'est quand OMEN fait appel à **enumPwd(η,l)** que les structures de données servant à générer les candidats sont réellement utilisées. Tout d'abord, la fonction doit calculer tous les vecteurs de la forme $a = (a_1, ..., a_{l-n+1})$. Ces vecteurs peuvent se calculer de manière récursive, et n'ont pas besoin d'être stockés car leur utilisation est indépendante. Si l'on prend $a = (a_1, ..., a_{l-n+1})$ l'un de ces vecteurs, OMEN construit un arbre de profondeur $l - n + 1$ où chaque niveau j de l'arbre correspond au choix d'un n-gram du niveau de probabilité a_j. La racine est étiquetée "^" indiquant le début du mot, tandis que le reste des nœuds sont étiquetés avec le n-gram choisi. Ainsi, l'ensemble des nœuds de profondeur 1 correspond à l'ensemble des n-grams de niveau a_1. La dernière condition pour construire cet arbre est que le préfixe de taille $n - 1$ de l'étiquette d'un nœud doit être le suffixe de taille $n - 1$ de l'étiquette de son père, afin de garantir que tout chemin de la racine à une feuille forme une chaîne de Markov. Générer les candidats de a s'effectue en parcourant cet arbre en profondeur, et en affichant un candidat à chaque fois que l'on visite une feuille, en effectuant une concaténation des étiquettes du chemin (avec chevauchement). Un exemple d'arbre se trouve sur la figure 6.13.

Coût de construction : pour construire un arbre de cette forme, il faut d'abord organiser les n-grams dans les niveaux de manière à pouvoir fabriquer des chaînes efficacement. Une bonne manière de faire est d'avoir un dictionnaire U pour chaque niveau η_i, où les clés sont les suffixes s de taille $n - 1$ et les valeurs une liste de n-grams ayant pour préfixe s et étant dans le niveau η_i. Ainsi, rechercher l'ensemble des n-grams de niveau a_i ayant pour préfixe le mot s se fait en temps constant en utilisant $U[a_i][s]$. Cette liste peut être vide, si aucun n-gram de niveau a_i ne commence par s (la conséquence est expliquée un peu plus bas). La construction de ce dictionnaire peut se faire pendant la phase d'apprentissage, durant l'étape de

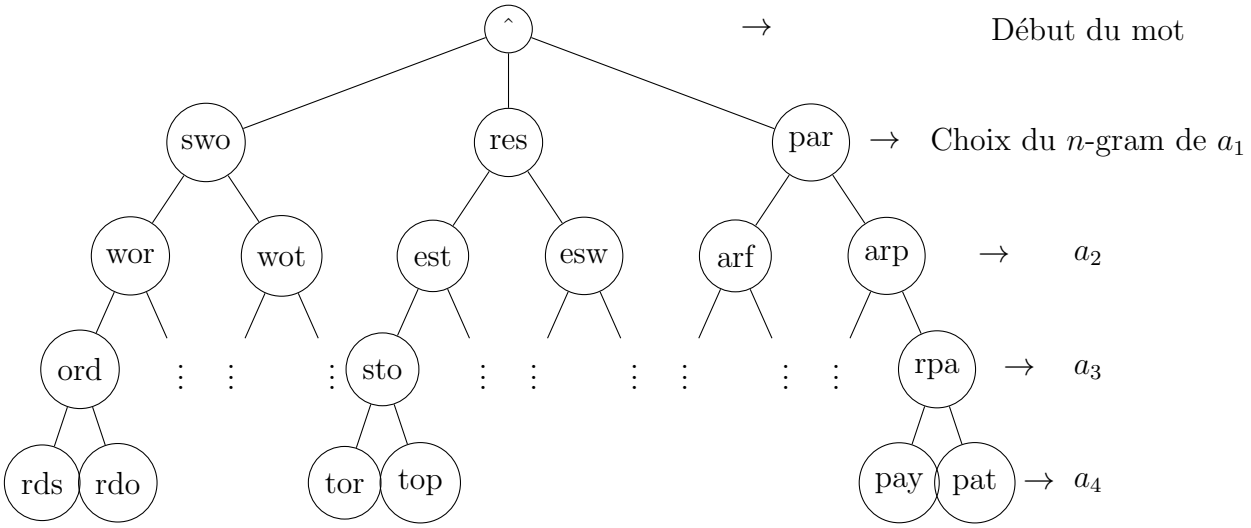


FIGURE 6.13 – Un exemple d'arbre. Ici les mots générés sont "swords", "swordo", ..., "restor", "restop", ..., "parpay", "parpat".

calcul des niveaux de chaque n-gram, sans coût supplémentaire en temps. Ensuite, étant donné un vecteur $a = (a_1, ..., a_{l-n+1})$, il est simple de parcourir en profondeur l'arbre associé. En pratique, il n'est pas nécessaire de construire explicitement cet arbre car il peut être parcouru à la volée en utilisant le dictionnaire créé à l'étape précédente. Pour cela, on utilise la fonction récursive définie par l'algorithme 1 où $pref$ est le début du prochain mot en cours de construction (servant de mémoire à l'appel récursif) et a le vecteur de niveaux de probabilités.

Algorithme 1 GénérerMots($pref = ""$, a)

```
η = a[0]
suffixe suf = pref[−(n − 1) :]          ▷ les n − 1 dernières lettres de pref
pour chaque n-gram g de U[η][suf] faire
    c = g[−1]                            ▷ la dernière lettre de g
    si taille(a) == 1 alors
        afficher pref + c                ▷ concaténation
    sinon
        GénérerMots(pref + c, a[1 :])
```

On appelle une première fois la fonction **GénérerMots** avec la chaîne vide comme préfixe et le vecteur a entier, puis à chaque appel récursif la fonction remplit le préfixe avec les n-grams sélectionnés et s'appelle à nouveau elle-même en retirant le premier élément de a. La taille des arbres ainsi générés dépend de plusieurs paramètres :

1. la longueur l des mots à générer.

   Les arbres seront de profondeur $l - n + 1$.

2. du nombre de n-gram de chaque niveau du vecteur a.

La taille de l'arbre est bornée par $\prod_{i=1}^{l-n+1} Card(\eta_i)$, le produit des cardinaux des niveaux de a. Cependant, toutes ces combinaisons ne forment pas une chaîne.

D'autre part, il n'est pas possible de savoir à l'avance en fonction de l, n, a et $Card(\eta_i)$ si l'on peut construire un tel arbre. En effet, il faut construire la racine et les premiers enfants avant de se rendre compte qu'il n'y a pas de n-gram pouvant continuer la chaîne entamée. Par exemple, imaginons que sur l'arbre en Figure 6.13, après avoir construit le nœud étiqueté "arp", il n'existe pas de 3-gram de niveau a_3 commençant par "rp", alors "arp" n'a pas de fils et donc cette partie de l'arbre ne sera pas de la bonne longueur. Mais ce cas ne peut pas être détecté avant, il est donc nécessaire de construire le début de l'arbre pour s'en rendre compte. Il arrive donc, qu'après avoir construit un grand nombre de nœuds, l'arbre ne permette pas de générer un seul candidat.

Nous allons donc mesurer le coût de génération des candidats en utilisant le nombre de nœuds d'arbre créés pour produire ce candidat.

6.3.4 Analyse du nombre de nœuds créés par candidat

Pour chaque candidat créé par l'algorithme de génération, nous allons mesurer le nombre de nœuds nécessaires pour le produire. Les bases de données utilisées sont Rockyou, LinkedIn et Myspace. Nous mesurons également, pour chaque candidat généré, le nombre de fois qu'il apparaît dans la base cible. De la même manière que nous avons fait avec PCFG, nous avons d'abord analysé le comportement de l'algorithme sur les 3 différentes bases de données sans application de filtre. Dans un deuxième temps, nous avons appliqué aux bases de données, un filtre qui interdit l'usage d'un n-gram au delà d'un certain nombre *seuil* d'utilisations. C'est à dire que si un n-gram est utilisé plus de *seuil* fois dans la base de données, aucun nouveau mot de passe le contenant ne sera autorisé. De la même manière que nous avons procédé avec PathWell, nous filtrons une base de la manière suivante : nous mélangeons la base de données, puis nous la parcourons pour compter l'utilisation de chaque n-gram. Si à tout moment un mot de passe contient un n-gram déjà utilisé plus de *seuil* fois, alors ce mot de passe n'est pas conservé. Nous appelons ce filtre "le filtre n-gram".

Nous avons choisi d'utiliser l'implantation en C d'OMEN fournie par les auteurs, car elle est suffisamment rapide et lisible pour y faire nos modifications. Nous avons également choisi d'utiliser les paramètres par défaut d'OMEN, en l'occurrence $n = 4$, car les expériences pour $n = 2$ et $n = 3$ n'ont pas permises d'apporter des résultats différents pour le nombre de nœuds générés par candidat. En revanche, $n = 4$ permet de trouver davantage de mots de passe dans la base de données que $n = 2$ et $n = 3$.

6.3.4.1 Choix des paramètres

Nous avons choisi de tracer la courbe du nombre de nœuds parcourus et le nombre de mots trouvés en fonction du nombre de candidats générés, c'est-à-dire en fonction de l'avancement de l'algorithme dans la génération des candidats. Nous

avons effectué ces mesures jusqu'à 10^{10} candidats car cela suffit pour mettre en évidence le comportement de l'algorithme, mais aussi car cela demande un temps d'exécution important.

Afin de rendre plus rapide la génération des candidats et de pouvoir aller assez loin dans l'énumération, nous n'avons pas récupéré le nombre de nœuds parcourus et le nombre de mots trouvés pour chaque candidat généré mais pour des blocs de 10^4 candidats. Ceci suppose donc que le nombre de nœuds parcourus et le nombre de mots trouvés sont constants sur ces 10^4 candidats. À l'échelle de l'énumération, pour laquelle nous sommes capables d'aller jusqu'à 10^{10} candidats, cette supposition n'influe pas sur les graphiques car c'est principalement le comportement sur une longue période que nous étudions ici.

Ensuite, malgré cette supposition, la variation du nombre de nœuds parcourus d'un bloc de 10^4 candidats à l'autre est trop importante pour que l'on puisse tracer un graphique lisible. Nous traçons donc à la place une moyenne sur les 10^4 blocs précédents le candidat généré. Ce n'est pas équivalent à calculer une moyenne sur 10^8 candidats car ici nous calculons une moyenne sur des blocs indépendants.

Dans la première expérience, nous avons effectué les mesures sur les bases non filtrées. Dans la deuxième expériences, nous avons effectué l'expérience avec $seuil = 10^4$ et $seuil = 10^3$. En fixant $seuil = 10^2$, nous avons constaté qu'OMEN devient terriblement lent à générer les candidats (2×10^8 après 20h d'énumération), ce qui veut dire qu'il passe son temps à construire des arbres pour générer très peu de candidats. Cette lenteur indique que ce genre de filtre est efficace pour se prémunir des attaques utilisant OMEN. En revanche, comme déjà énoncé dans ce manuscrit, nous n'avons pas de garantie sur l'utilisabilité d'une telle méthode.

6.3.4.2 Résultats sur les bases non filtrées

Sur LinkedIn non filtrée (Figure 6.14), OMEN parcourt au début de l'énumération en moyenne 10^6 nœuds pour générer 10^4 candidats, et ceci jusqu'à environ 0.7×10^{10} candidats. Par la suite, ce nombre varie entre 10^5 et 10^6 nœuds. On remarque que le nombre de nœuds parcourus n'augmente pas au fil de l'énumération, il a même parfois tendance à baisser. Concernant le nombre de mots trouvés par bloc de 10^4 candidats, ce nombre oscille entre 1 et 10^3 au tout début de l'énumération, et devient nul entre 0.2×10^{10} et 0.7×10^{10}.

Sur Rockyou non filtrée (Figure 6.15), on remarque un comportement similaire à celui sur LinkedIn. La moyenne du nombre de nœuds parcourus oscille autour de 5×10^5 jusqu'à 0.5×10^{10} candidats. Au delà, ce nombre montre une plus grande variation entre 10^5 et 10^6. La courbe du nombre de mots trouvés possède la même allure que celle sur LinkedIn.

Sur Myspace non filtrée (Figure 6.16), la moyenne du nombre de nœuds parcourus pour la génération des candidats ne varie pas beaucoup durant l'énumération, il oscille entre 5×10^5 et 7×10^5. On remarque cependant que l'allure de la courbe du nombre de mots trouvés durant l'énumération est caractéristique du fonctionnement d'OMEN, c'est-à-dire qu'il génère des mots par longueur l (par exemple $l = 3$ puis $l = 4$ puis $l = 5$), et pour chaque longueur il parcourt des ensembles de mots qu'il

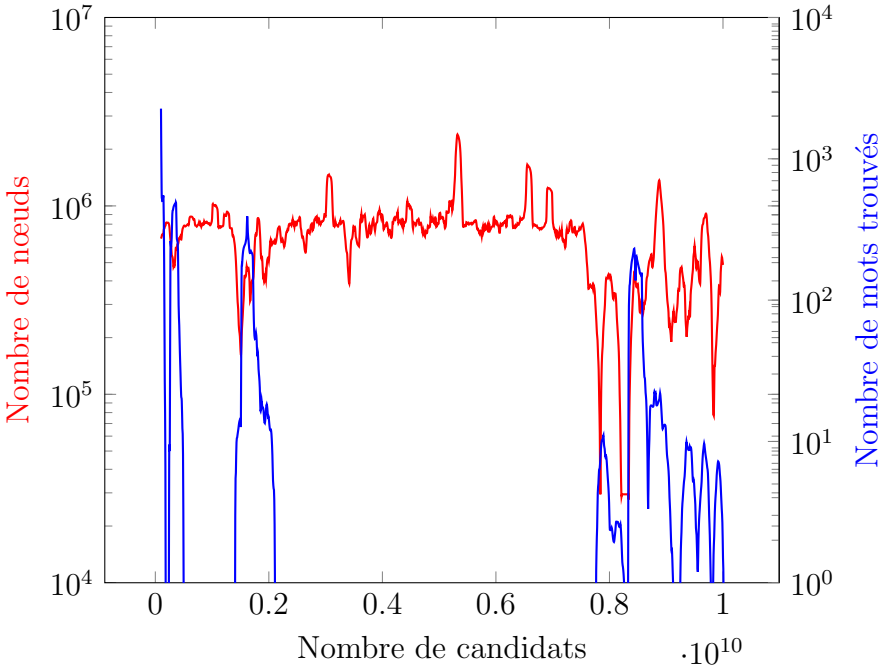


FIGURE 6.14 – OMEN sur LinkedIn

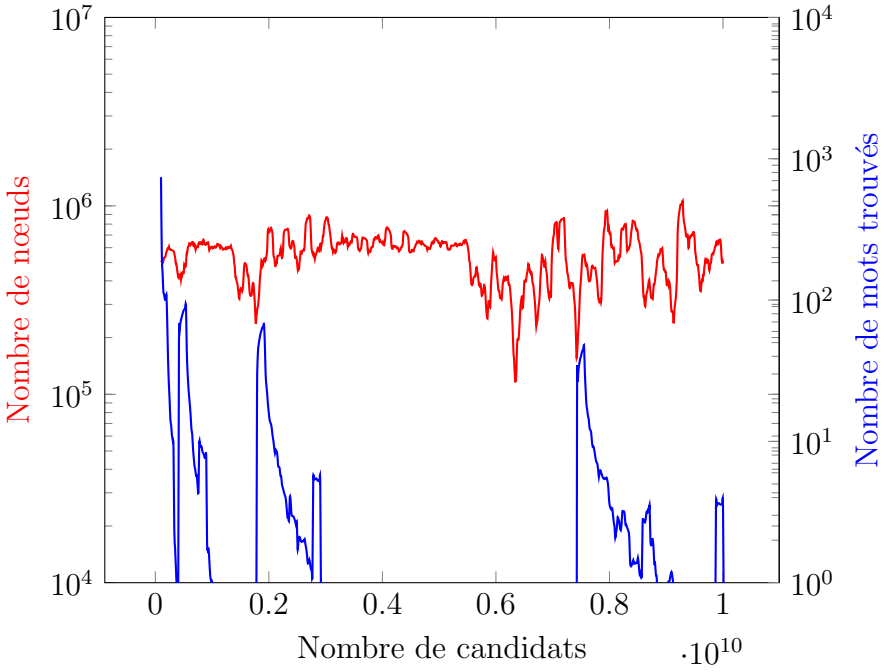


FIGURE 6.15 – OMEN sur Rockyou

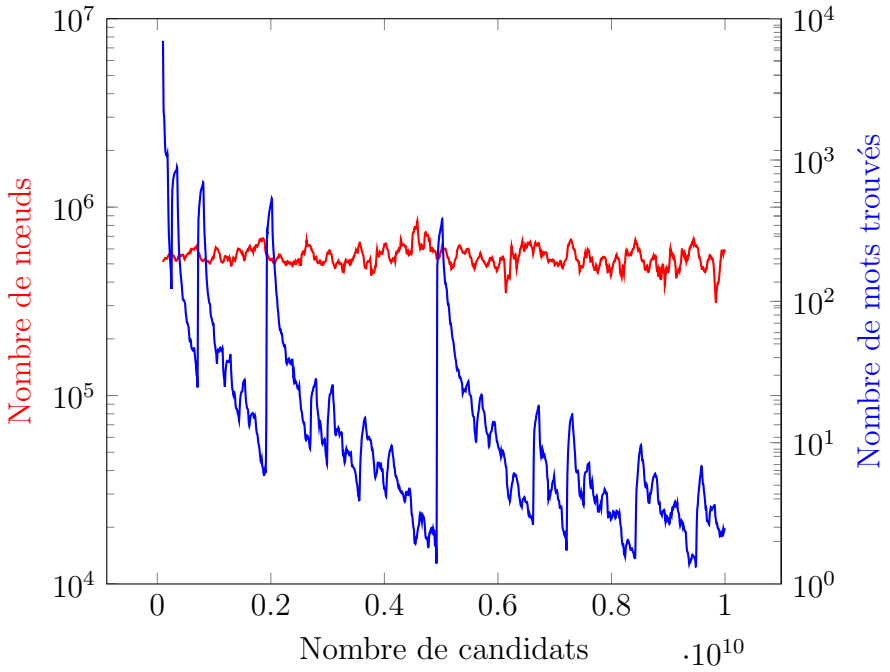


FIGURE 6.16 – OMEN sur Myspace

ordonne par probabilité décroissante. Ce qui explique la forme de "montée subite puis descente en oscillant". Les montées subites correspondent au changement de longueur de mot, tandis que chaque oscillation correspond à un ensemble de mots pour un vecteur a donné. On remarque également que les courbes du nombre de mots trouvés sur LinkedIn et Rockyou sont de la même forme que celle sur Myspace sauf qu'elles sont plus faibles d'un ordre de grandeur.

6.3.4.3 Résultats sur les bases filtrées avec le filtre n-gram

Lorsque l'on attaque LinkedIn où un seuil de 10^4 est utilisé pour le filtre n-gram (Figure 6.17), on remarque que la moyenne du nombre de nœuds parcourus oscille beaucoup plus que sans filtre. On aperçoit à la fois des pics bas à 3×10^4 nœuds, ce qui indique en moyenne 3 nœuds parcourus pour générer un candidat, à la fois des pics hauts pouvant aller jusqu'à 10^7 nœuds. Cependant, on ne voit pas d'augmentation claire du nombre de nœuds parcourus sur l'ensemble de l'énumération. En effet, à certaines périodes ce nombre augmente effectivement, par exemple entre 0.4×10^{10} et 0.8×10^{10}, mais décroît fortement à d'autres périodes, par exemple entre 0.2×10^{10} et 0.35×10^{10} candidats.

Le nombre de mots trouvés est quant à lui bien inférieur à celui de LinkedIn sans filtre, ce qui est rassurant sur le fait que le filtre n-gram est une méthode pertinente pour réduire la performance de l'attaque avec OMEN, en tout cas sur LinkedIn.

Le seuil à 10^3 (Figure 6.18) permet de confirmer les explications pour le seuil à 10^4, même si l'on remarque une bien plus grande période de pic bas entre 0.5×10^{10} et 0.9×10^{10} candidats. Le nombre de mots trouvés est quant à lui encore plus faible.

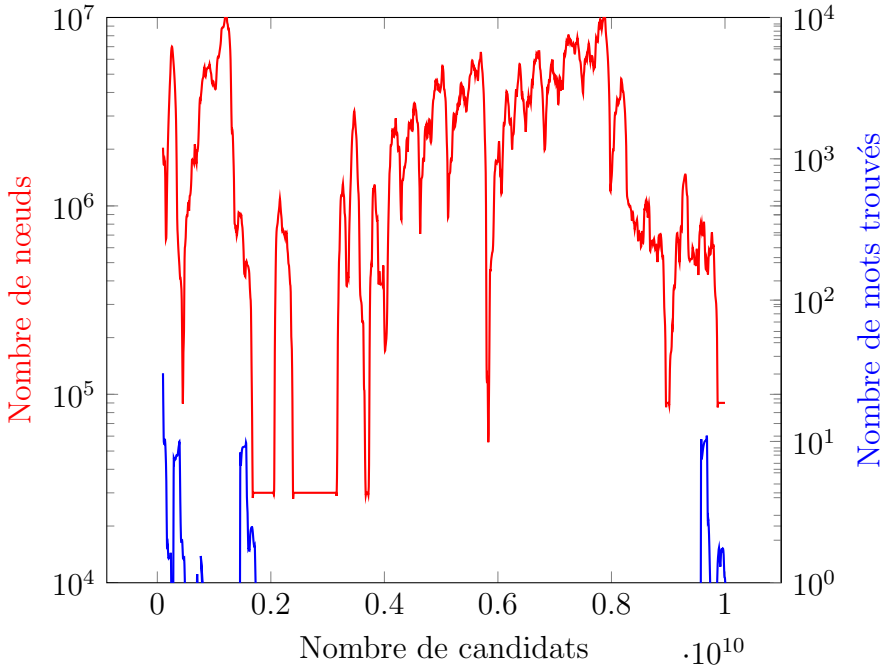


FIGURE 6.17 – OMEN sur LinkedIn avec *seuil* = 10^4

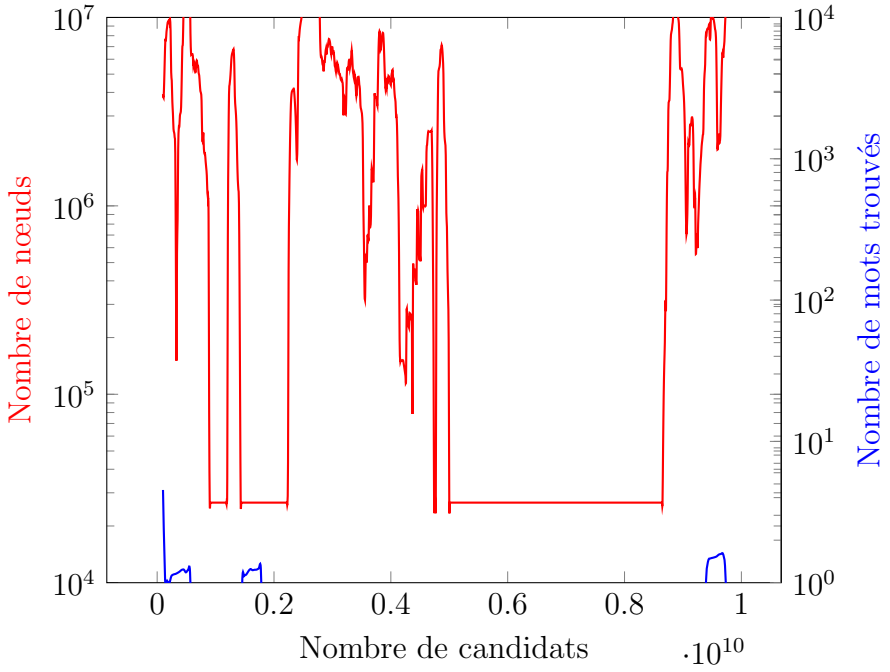


FIGURE 6.18 – OMEN sur LinkedIn avec *seuil* = 10^3

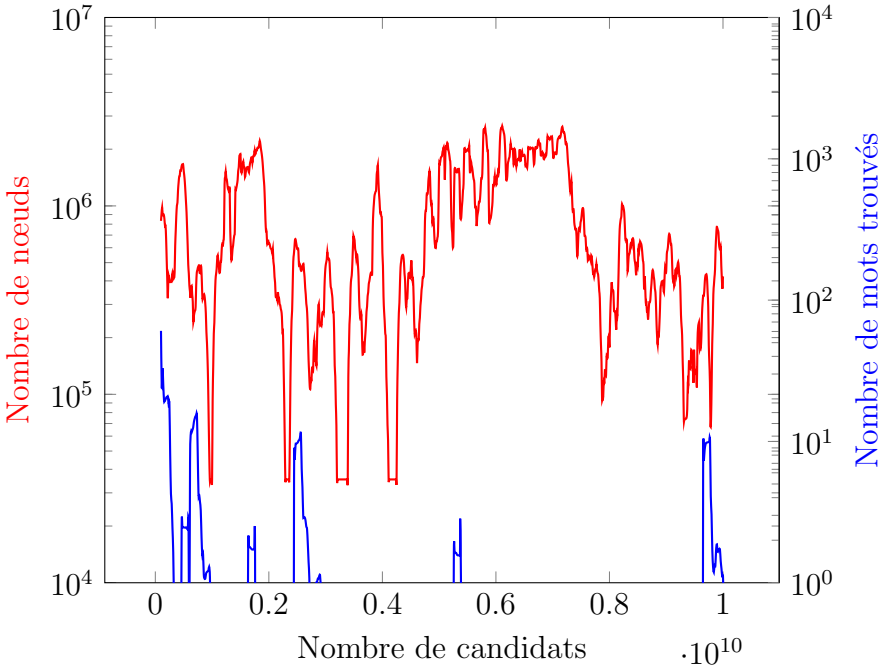


FIGURE 6.19 – OMEN sur Rockyou avec *seuil* = 10^4

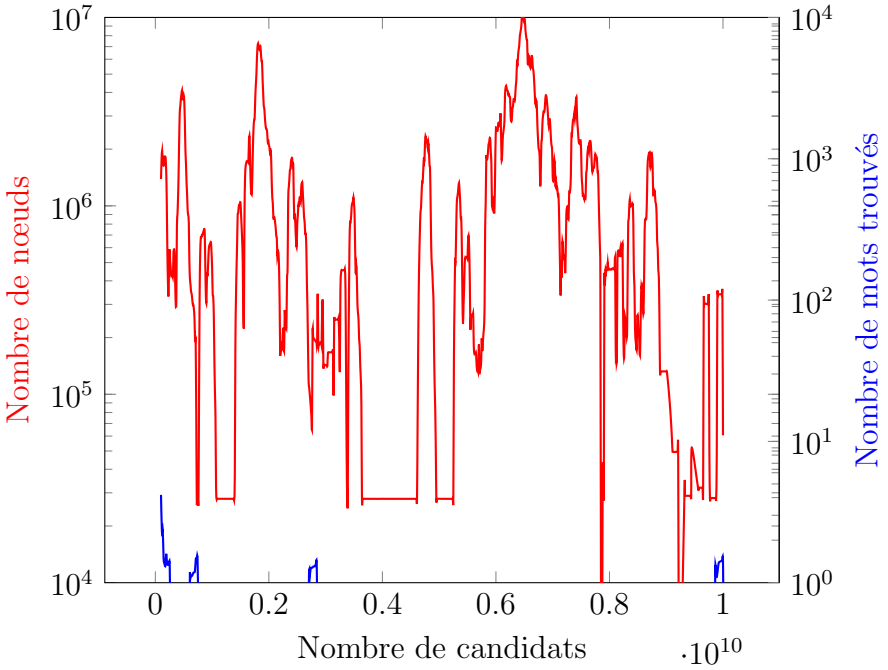


FIGURE 6.20 – OMEN sur Rockyou avec *seuil* = 10^3

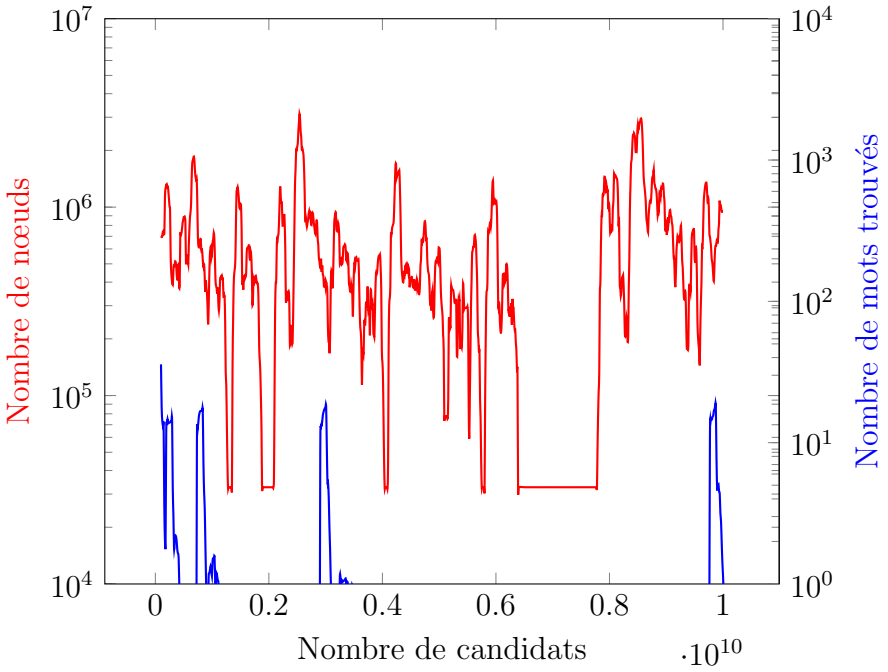


FIGURE 6.21 – OMEN sur Myspace avec $seuil = 10^4$

Sur Rockyou, on remarque que paramétrer le filtre n-gram avec $seuil = 10^4$ (Figure 6.19) ne permet pas de grandement augmenter le nombre moyen de nœuds parcourus, contrairement à $seuil = 10^3$ (Figure 6.20) où l'on peut observer des pics hauts à 10^7 plus fréquemment. Même si avec $seuil = 10^4$ le nombre moyen de nœuds oscille davantage que sans filtre, les pics hauts ne sont que 2 à 3 fois plus élevés que sans filtre, tandis que les pics bas sont 2 fois plus bas avec $seuil = 10^4$ que sans filtre. Le nombre de mots trouvés est également grandement réduit lorsque l'on filtre Rockyou avec le filtre n-gram.

Là où sans filtre la courbe du nombre moyen de nœuds parcourus était relativement stable sur Myspace, ce n'est plus le cas lorsqu'elle est filtrée avec le filtre n-gram paramétré à $seuil = 10^4$ (Figure 6.21) et $seuil = 10^3$ (Figure 6.22). En effet, la courbe oscille beaucoup plus avec $seuil = 10^4$ et encore plus avec $seuil = 10^3$. Là où avec $seuil = 10^4$ la courbe oscille autour de la même valeur que sans filtre, avec $seuil = 10^3$ la courbe oscille plutôt autour de 2×10^6, soit 4 fois la valeur sans filtre. Le nombre de mots trouvés est encore une fois très faible indiquant que le filtre n-gram permet de réduire drastiquement les performance d'OMEN sur ces trois bases de données.

6.4 Conclusion

Nous avons présenté dans ce chapitre une analyse du comportement de deux algorithmes d'énumération basés sur des modèles probabilistes : l'algorithme de la grammaire de Weir et OMEN.

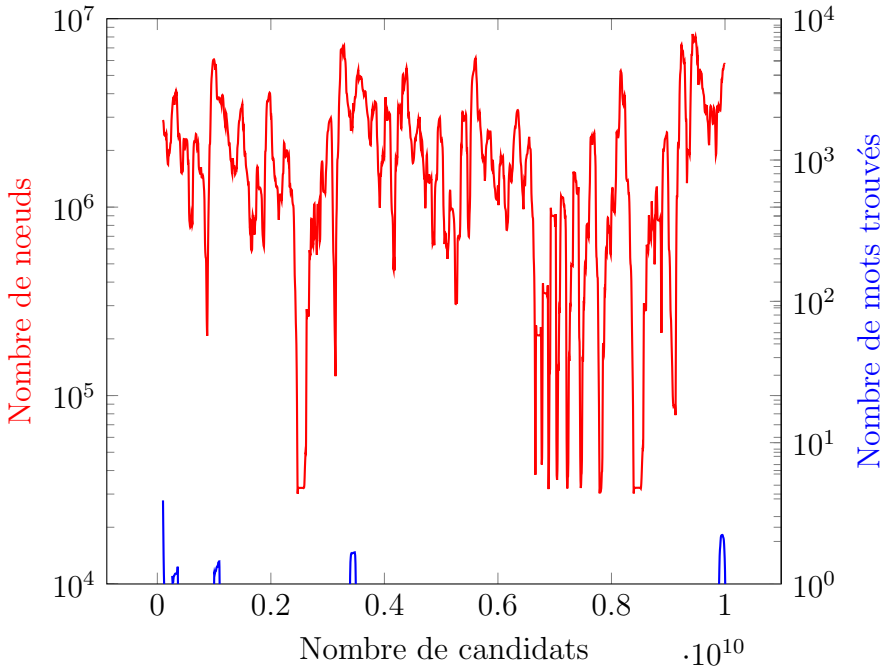


FIGURE 6.22 – OMEN sur Myspace avec $seuil = 10^3$

Nous proposons une nouvelle manière de mesurer le coût de génération des candidats pour les énumérateurs PCFG et OMEN. Cette manière d'analyser le coût de génération des candidats permet de mieux comprendre comment fonctionnent ces algorithmes, rendant plus simple la conception de défenses pour se protéger des attaques les utilisant. Cette méthode d'analyse du coût de génération est complémentaire à la mesure du temps de génération d'un candidat utilisée dans le chapitre 4.

Pour PCFG, nous avons remarqué que le coût de génération d'un candidat était principalement lié aux opérations effectuées sur la file de priorité de l'algorithme. Cette structure de données est unique tout de long de l'algorithme, ce qui rend assez facile son analyse. Ceci permet de confirmer le fait que plus on va loin dans l'énumération des candidats, plus il devient coûteux de les générer, même si ce coût n'est pas linéaire en le nombre de candidats générés. Il faudrait pour cela être capable de faire grandir la file de priorité de manière exponentielle, en obligeant l'algorithme à ajouter toujours de plus en plus d'éléments dans la file. Ceci est possible lorsque le nombre de bases simples des éléments de la file augmente au fil de l'énumération, ce qui demande à ajouter 1 élément à la file au début, puis 2, puis 3, 4, ... pour chaque candidat généré. De cette façon, la taille de la file grossirait de manière exponentielle.

Les expériences montrent également que **PathWell** est une bonne méthode pour se prémunir des attaques utilisant PCFG, au moins du point de vue de la performance de l'attaque, car elle permet d'augmenter le coût de génération des candidats, en faisant croître davantage la taille de la file, mais aussi en réduisant significativement le nombre de mots trouvés. PathWell n'est pas une méthode compliquée à mettre en place, même si une seule publication montre qu'elle est bien acceptée par les utilisateur [45].

OMEN est cependant plus difficile à analyser algorithmiquement que PCFG, car il n'utilise pas une unique structure de données mais parcourt des arbres différents tout au long de l'énumération. De plus, le nombre de nœuds d'un arbre dépend de plusieurs paramètres qui ne sont pas tous prévisibles tel que le nombre de n-grams d'un niveau fixé commençant par une préfixe donné. Contrairement à PCFG, cette mesure du coût de génération des candidats ne permet pas de montrer que ce coût augmente durant l'énumération. Ainsi, il ne semble pas réellement plus difficile pour OMEN de générer des candidats après 5×10^9 candidats qu'au début de l'énumération, même s'il existe des périodes durant lesquelles la difficulté varie significativement. Mesurer le nombre de nœuds parcourus pour générer un candidat n'est peut-être pas la meilleure méthode pour mettre en avant l'augmentation du coût de génération des candidats durant l'énumération, mais reste tout de même une bonne méthode pour montrer la différence de coût entre les versions non filtrées et filtrées des bases de données.

Le filtre n-gram permet d'augmenter le coût de génération des candidats pour OMEN, ainsi que de faire baisser drastiquement le nombre de mots trouvés, ce qui montre que c'est une bonne méthode pour faire baisser les performances d'OMEN. Cependant, ce filtre semble plus contraignant que PathWell car bannir un n-gram a probablement plus d'impact sur les nouveaux mots de passe que bannir une base composée.

PathWell et le filtre n-gram ne sont probablement pas les meilleures manières de faire baisser les performances de PCFG et OMEN, elles semblent cependant suffisantes et efficaces. En revanche, même si PathWell est montré comme étant appréciée des utilisateurs [45], notre filtre n-gram le sera probablement moins.

Une meilleure alternative au filtre n-gram serait de s'inspirer de [7] où les auteurs comptent le nombre d'occurrences de chaque n-gram et rejettent les mots de passe qui utilisent trop de n-grams trop populaires. Pour cela, on calcule la probabilité d'apparition $p(w_i)$ de chaque n-gram w_i rencontré dans la base de données, et on fixe $p(w_i) = 0$ pour les w_i non rencontrés dans la base de données. Quand un mot de passe est entré dans le système, on calcule un score

$$s(w = w_1|...|w_{l-n+1}) = \prod_{i=1}^{l-n+1} p(w_i)$$

et on accepte w comme mot de passe si $s(w) \geq t$ où t est un seuil de score déterminé à l'avance. Ainsi on ne se base plus sur la popularité d'un seul n-gram mais sur la popularité de l'ensemble des n-grams du mot.

Chapitre 7

Piste pour la mise en œuvre d'une mesure de robustesse efficace

Dans ce chapitre, nous synthétisons les travaux des chapitres précédents dans le but de proposer des pistes de réflexion pour la mise en œuvre de mesures de robustesse efficaces contre les énumérateurs probabilistes. On se place dans la situation où l'on souhaite mesurer la robustesse d'un mot de passe en fonction d'un ensemble de mots de passe. Dans un premier temps, l'objectif est de définir formellement les propriétés souhaitées d'une mesure de robustesse pour qu'elle soit efficace dans le temps. En même temps que de chercher à définir une mesure de robustesse qui soit universelle, il faut également mettre en place des solutions plus pratiques telles que les métriques multimodales de robustesse. Ensuite, nous proposons des suggestions d'amélioration aux mesures existantes pour prendre en compte les attaques probabilistes.

Sommaire

7.1 Propriétés souhaitées d'une mesure de robustesse

Dans cette partie, nous définissons les propriétés que l'on souhaite satisfaire lorsque l'on conçoit une mesure de robustesse moderne, qui soit précise et pertinente pour se prémunir des attaques sur les mots de passe.

Dans leur publication en deux parties [15, 16], Javier Galbally, Iwen Coisel et Ignacio Sanchez identifient les propriétés nécessaires à une mesure de robustesse des

mots de passe pour être efficace dans le temps. Ils exposent dans un premier temps les 3 catégories de mesures de robustesse existantes dans la littérature :

- **catégorie 1 :** robustesse basée sur les attaques. Les métriques mesurent la robustesse des mots de passe en fonction du temps nécessaire aux attaques connues pour les casser.
- **catégorie 2 :** robustesse basée sur des heuristiques. Ces métriques mesurent la robustesse de manière intrinsèque au mot de passe, par exemple en fonction du nombre de caractères et/ou des classes de chacun. La mesure proposée par le NIST [38] en est un exemple.
- **catégorie 3 :** robustesse basée sur des modèles probabilistes. Ces métriques mesurent la robustesse des mots de passe en fonction de statistiques recueillies sur des bases de données fuitées de mots de passe. Les exemples typiques sont les chaînes de Markov et les grammaires probabilistes.

Chacune de ces catégories a ses avantages et ses inconvénients.

- **Catégorie 1 : robustesse basée sur une attaque**
  - **Avantages :** Ces méthodes permettent d'estimer le temps nécessaire à une ou plusieurs attaques pour casser les mots de passe, permettant d'avoir une vision pratique et temporelle de la résistance des mots de passe. Elles prennent en compte la puissance de calcul de l'attaquant, et permettent donc de s'y adapter.
  - **Inconvénients :** Il faut prendre en compte toutes les attaques connues sur les mots de passe afin de pouvoir proposer une méthode pertinente pour le plus de mots de passe possibles. De plus, ces méthodes ne permettent pas de prendre en compte le contexte d'utilisation du mot pour proposer une robustesse plus précise.
- **Catégorie 2 : robustesse basée sur des heuristiques**
  - **Avantages :** Ces méthodes sont très rapides à exécuter, car elles ne demandent aucune ressource externe ni calcul coûteux. Elles permettent globalement de se protéger des attaques les plus naïves telles que la force brute.
  - **Inconvénients :** Ces méthodes ne prennent pas en compte les biais lors de la création des mots de passe par les utilisateurs. On sait que le choix d'un mot de passe par un humain n'est pas aléatoire, et donc ces méthodes ne sont pas efficaces si les modifications réalisées pour satisfaire les heuristiques sont trop prédictibles. De plus, ces méthodes génèrent bien souvent de la frustration auprès des utilisateurs car elles demandent en général plus de tentatives pour passer les règles.
- **Catégorie 3 : robustesse basée sur des modèles probabilistes**
  - **Avantages :** Ces méthodes prennent en compte les biais des utilisateurs lors du choix de leur mot de passe, en calculant des statistiques sur l'utilisation de motifs, de répétitions, sur un ensemble de mots de passe. Ainsi, un mot de passe qui contient un motif trop souvent rencontré dans

les mots de passe ayant fuités verra sa robustesse baisser. Ces méthodes permettent également d'augmenter la résistance des mots de passe face aux attaques plus sophistiquées telles que les grammaires probabilistes ou les modèles de chaînes de Markov.

— **Inconvénients :** Elles ont besoin d'entraîner les modèles sur des données réelles, il faut donc avoir des données suffisamment représentatives et de taille suffisamment conséquente. De plus, les modèles entraînés occupent relativement beaucoup de place en mémoire, ce qui demande des ressources plus importantes pour les utiliser. Il est par exemple quasiment inenvisageable d'utiliser ces méthodes dans le navigateur (en Javascript par exemple), et le calcul doit se faire côté serveur.

Des avantages et inconvénients de chaque méthode, les auteurs en font deux conclusions :

1. il n'existe pas de méthode universelle, unique, permettant de mesurer la robustesse des mots de passe de manière pleinement satisfaisante ;
2. les mesures de robustesse doivent s'adapter aux différents environnements afin de proposer des mesures précises.

On appelle ici environnement l'ensemble des informations contextuelles dans lesquelles évoluent les mots de passe. Il peut s'agir à la fois d'informations générales telles que la langue du site web, le public visé par le site, l'importance apportée à la sécurité du compte par les utilisateurs (un jeu vidéo a peut-être moins de valeur qu'une boîte mail ou un profil de réseau social), et à la fois d'informations plus spécifiques telles que la politique de création de mot de passe et la base de données (afin de faire de l'apprentissage).

Ce qui les amènent à proposer deux objectifs pour les nouvelles mesures de robustesse à concevoir :

1. les nouvelles mesures doivent être multimodales, c'est-à-dire qu'elles doivent exploiter les avantages de différentes méthodes afin de couvrir le plus de menaces possibles envers les mots de passe.
2. les mesures doivent pouvoir s'adapter à l'environnement dans lequel évoluent les mots de passe dont on souhaite mesurer la robustesse, grâce à un processus d'entraînement.

Nous proposons d'ajouter un volet mise à jour à l'adaptabilité d'une métrique de robustesse. En effet, si au lancement d'un service la métrique de robustesse est pertinente et bien paramétrée (langue, disposition du clavier, entraînement sur des données proches de la langue, ...), cela peut évoluer avec le temps et donc ne plus être vrai dans le futur. Après quelques années et des centaines de milliers de mots de passe entrés dans le système, on peut retrouver beaucoup de comportements récurrents (chiffres à la fin, mots du dictionnaire anglais, ...), ce qui n'est pas souhaitable. Nous proposons donc de demander qu'une métrique de robustesse puisse évoluer avec le temps, en mettant à jour ses paramètres et de refaire son entraînement. Évidement, ce processus peut être fait manuellement en relançant l'apprentissage des modèles probabilistes par exemple, mais c'est très coûteux et bien souvent impossible car

les mots de passe ne sont pas stockés en clair. Il est alors intéressant d'ajouter une étape de mise à jour des modèles dans la métrique de robustesse.

À ces deux objectifs, nous rajouterons un troisième essentiel pour nous, qui est l'utilisabilité. En effet, même si une mesure de robustesse est multimodale et s'adapte très bien au contexte des mots de passe, si cette mesure n'est pas facile à utiliser alors elle ne sera pas adoptée car trop frustrante.

Ces objectifs se regroupent dans deux notions : la sécurité, pour la multimodalité et l'adaptabilité, et l'utilisabilité.

Si l'on résume, on souhaite atteindre 3 objectifs lors de la conception d'une métrique de robustesse des mots de passe :

— multimodalité ;

— adaptabilité au contexte et mise à jour des paramètres ;

et un objectif secondaire : l'utilisabilité.

Il est fort probable que ces 3 objectifs ne puissent pas être atteints de manière très satisfaisante pour une même métrique de robustesse. De plus, l'utilisabilité est la variable d'ajustement car on souhaite d'abord fixer le curseur de la sécurité (multimodalité et adaptabilité) et ensuite avoir une valeur d'utilisabilité la plus grande.

Lorsque l'on conçoit une mesure de robustesse des mots de passe, on souhaite qu'elle permette de renforcer la sécurité des mots de passe de manière significative mais également qu'elle suscite le moins de frustration possible. Nous avons donc un système où l'on souhaite maximiser deux fonctions : la sécurité et l'utilisabilité. Cependant, nous estimons que la sécurité est plus importante que l'utilisabilité.

7.1.1 Multimodalité

Puisqu'il n'est pas possible de trouver une unique mesure de robustesse qui soit efficace pour toutes les attaques, il faut proposer une métrique de robustesse qui rassemble plusieurs métriques élémentaires afin d'obtenir les avantages de chacune d'entre elles. C'est ce qu'on appelle une métrique multimodale.

Définition : Soient $s_1, s_2, ..., s_k$ des métriques élémentaires de robustesse des mots de passe, c'est-à-dire que chaque s_i donne un score à w vis-à-vis d'une attaque fixée a_i de n'importe quel type (force brute, dictionnaire, modèle probabiliste, ...). Une métrique multimodale est une fonction $S : \mathbb{R}^k \to \mathbb{R}$ prenant en entrée k scores de robustesse et calculant un score de robustesse multimodale.

Exemple 1 :

$$S(w) = min(s_1(w), s_2(w), ..., s_k(w))$$

est un exemple simple de mesure de robustesse multimodale qui consiste à supposer que la robustesse de w est sa robustesse face à l'attaque la plus efficace pour le casser. Ici, on se place dans le modèle d'un attaquant qui choisit directement l'attaque la plus efficace contre le mot de passe w. C'est une mesure de robustesse conservative, qui offre une borne peut-être trop inférieure.

Exemple 2 :

$$S(w) = \sum_{i=1}^{k} \frac{s_i(w)}{k}$$

est une mesure multimodale consistant à calculer la moyenne des scores de robustesse donnés par les métriques élémentaires. Ici, on se place dans le modèle d'un attaquant qui choisi uniformément une attaque dans $\{a_1, ..., a_k\}$ pour casser w.

Exemple 3 :

$$S(w) = \sum_{i=1}^{k} \frac{\pi_i.s_i(w)}{\Pi}$$

où π_i est la pondération affectée à s_i et où $\Pi = \sum_{i=1}^{k} \pi_i$. Ici, on pondère le score donné par chaque s_i car on se place dans le modèle d'un attaquant qui privilégie certaines attaques, en considérant que chaque attaque n'a pas le même impact sur la robustesse. Par exemple, si l'attaque a_1 est moins coûteuse que les autres, on peut lui affecter une pondération plus élevée car elle sera probablement utilisée plus souvent par l'attaquant.

On peut s'inspirer des travaux réalisés en biométrie concernant la fusion d'information [24], comme le suggèrent les auteurs de [15, 16]. Ils utilisent dans leur publication une somme pondérée pour fusionner les scores de plusieurs métriques de robustesse.

7.1.2 Adaptabilité au contexte et mise à jour des paramètres

Une bonne métrique de robustesse doit pouvoir s'adapter au contexte dans lequel les mots de passe vivent et également pouvoir évoluer au cours du temps.

7.1.2.1 Adaptabilité

Définition : Soit $\mathcal{W}$ l'ensemble des mots de passe d'un service donné. On note $C = c_1, c_2, ..., c_n$ le contexte constitué d'un ensemble d'éléments de diverses natures (la langue, la disposition du clavier, ...). On note $S(w|C)$ la robustesse du mot w quand S connaît le contexte, et $S(w)$ quand elle ne le connaît pas. On dit que S est adaptable si

$$\exists w \in \mathcal{W} | S(w|C) \neq S(w)$$

c'est-à-dire qu'au moins un mot de la base de l'ensemble $\mathcal{W}$ a vu son score de robustesse évoluer. Autrement dit, S prend en compte le contexte C dans l'évaluation de la robustesse de w.

En général, la connaissance de C apporte de l'information à S, c'est-à-dire que

$$\nexists w \in \mathcal{W} | S(w|C) > S(w).$$

En revanche, il peut exister des situations où connaître du contexte est contre-productif, par exemple lorsqu'un mot de passe dérivé de la langue française est enregistré dans une base de données principalement anglaise. Dans ce cas, connaître

le fait que la base de données soit en anglais peut ralentir l'attaquant. Cependant, on suppose ici que le contexte est pertinent.

Définition : On dit que S est progressive en connaissance de C si

$$\sum_{w \in \mathcal{W}} \frac{S(w|C)}{\#\mathcal{W}} < \sum_{w \in \mathcal{W}} \frac{S(w)}{\#\mathcal{W}}$$

c'est-à-dire que la connaissance de C permet à S d'être plus précis sur l'évaluation de la robustesse en moyenne des mots de $\mathcal{W}$.

On souhaite donc qu'une mesure de robustesse soit progressive en connaissance de C. On souhaite également trouver le contexte C qui permette à S d'être la plus progressive possible.

7.1.2.2 Mise à jour des paramètres

Définition : Soient $\mathcal{W}$ l'ensemble des mots de passe d'un service à un instant t, et $\mathcal{W}'$ l'ensemble des mots de passe à un instant t', avec $t' > t$ et $\#\mathcal{W}' > \#\mathcal{W}$. On note $S^t(w)$ la robustesse du mot w à l'instant t. On dit que S a été mise à jour entre t et t' si

$$\exists w \in \mathcal{W} | S^t(w) \neq S^{t'}(w)$$

c'est-à-dire qu'au moins un mot de $\mathcal{W}$ a vu sa robustesse évoluer entre t et t'. Autrement dit, S a pris en compte les nouveaux mots de passe, ceux de l'ensemble $\mathcal{W}' - \mathcal{W}$, pour ajuster la robustesse des anciens mots de passe.

Cette mise à jour peut se faire par exemple en relançant la phase d'apprentissage des modèles probabilistes, ce qui est très coûteux et impossible en pratique car les mots de passe ne sont pas stockés en clair. Au lieu de relancer l'apprentissage complet à chaque ajout de mot de passe, on peut mettre à jour le modèle uniquement en ayant le nouveau mot de passe w. Dans le cas de PCFG, il faut calculer sa base composée $\mathcal{Q}_w = B_1...B_k$ ainsi que les terminaux $T_1...T_k$ correspondants, et mettre à jour les probabilités dans le modèle. Il est nécessaire de conserver le nombre d'occurrences de chaque base composée et de chaque terminal afin de pouvoir refaire les calculs de probabilité.

Mise à jour de p_S. Soit q le nombre de bases composées enregistrées dans le système, $\mathcal{Q}_w$ la base composée du nouveau mot w et $p_S(\mathcal{Q}_w)$ la probabilité de $\mathcal{Q}_w$. Il faut ajuster la probabilité des bases composées. Pour chaque base composée $\mathcal{Q}_i$ du système, on a

$$p'_S(\mathcal{Q}_i) = \frac{b + q.p_S(\mathcal{Q}_w)}{1 + q}$$

où

$$b = \begin{cases} 1 & \text{si } \mathcal{Q}_i = \mathcal{Q}_w, \\ 0 & \text{sinon.} \end{cases}$$

Mise à jour de p_{B_i}. Soit T_i le terminal correspondant à la base simple B_i dans le mot $w = T_1...T_k$. Soit t le nombre de terminaux de la base simple B_i enregistrés

dans le système. Pour chaque terminal T_j de la base simple B_i enregistrés dans le système, on calcule la nouvelle probabilité de B_i telle que

$$p'_{B_i}(T_j) = \frac{b + t.p_{B_i}(T_j)}{1 + t}$$

où

$$b = \begin{cases} 1 & \text{si } T_i = T_j, \\ 0 & \text{sinon.} \end{cases}$$

7.1.3 Utilisabilité

L'utilisabilité U d'une métrique de robustesse est sa facilité d'utilisation. Elle est inversement proportionnelle à la frustration engendrée par l'utilisation de cette métrique. L'utilisabilité peut se mesurer de plusieurs manières.

7.1.3.1 Nombre moyen de tentatives

La première manière consiste à mesurer le nombre moyen de tentatives des utilisateurs pour faire valider leur mot de passe.

Définition : Soient $w_1, ..., w_m$ les mots de passe enregistrés dans un système donné. On note n_i le nombre de tentatives pour que le mot de passe w_i soit accepté dans le système. L'utilisabilité U_M en nombre moyen de tentatives de la métrique S est définie telle que

$$U_M(S) = \frac{1}{m} \sum_{i=1}^{m} n_i.$$

Plus $U_M(S)$ est grand, moins S est utilisable. On peut penser cependant que faire beaucoup de tentatives n'est pas toujours frustrant pour un utilisateur.

7.1.3.2 Proportion d'abandons

La deuxième méthode consiste à mesurer la proportion d'abandons de création de compte sur le système.

Définition : Soient q le nombre de fois où les utilisateurs ont abandonné le processus de création de compte, et m le nombre de mots de passe dans la base de données. L'utilisabilité U_A en proportion d'abandons de la métrique S est définie telle que

$$U_A(S) = \frac{q}{m}.$$

Plus $U_A(S)$ est grand, moins S est utilisable. On remarque que $U_A(S)$ peut être plus grand que 1 car il peut y avoir plus d'utilisateurs ayant abandonnés que d'utilisateurs s'étant enregistrés.

7.2 Suggestions d'amélioration de ZXCVBN

ZXCVBN est un bon exemple de métrique de robustesse multimodale, que nous avons présenté en section 3.3.1.1. Cette métrique permet de prendre en compte plusieurs attaques possibles sur les mots de passe, en découpant le mot en jetons selon des règles pré-établies et en mesurant la robustesse de chaque jeton face à chaque attaque supportée. Cette mesure est également adaptable au contexte d'utilisation du mot de passe car le dictionnaire utilisé, la puissance de calcul de l'attaquant, la langue dans laquelle est construit le mot de passe, la disposition du clavier sont autant de paramètres de cette métrique. Cependant, elle ne s'adapte pas automatiquement au fur et à mesure que les mots de passe sont entrés, c'est-à-dire qu'aucune étape de mise à jour des paramètres et des ressources n'est prévue, ce qui est pourtant ce que nous pouvons souhaiter d'une métrique de robustesse.

L'inconvénient de cette métrique, comme exposé dans le chapitre 5, est qu'elle n'est pas efficace pour se protéger des attaques basées sur les modèles probabilistes. On ne peut donc pas recommander son utilisation aujourd'hui. C'est pourquoi nous proposons des améliorations afin qu'elle devienne efficace contre ces attaques.

7.2.1 Intégrer les modèles probabilistes

Puisque cette métrique est multimodale, il est possible d'y ajouter des modules pour prendre en compte ces attaques basées sur les modèles probabilistes. Par exemple, en entraînant une PCFG, on peut ainsi donner un score

$$S(w) = p_{\mathcal{S}}(\mathcal{Q}) \prod_{i=1}^{k} P_{B_i}(T_i)$$

à un mot $w = T_1...T_k$ tel que défini dans la formule 6.1. ZXCVBN calcule un nombre d'essais moyen nécessaires pour trouver un jeton, il faut alors transformer $S(w)$ en un nombre d'essais pour l'intégrer à ZXCVBN. On peut par exemple utiliser une méthode de Monte-Carlo telle que définie dans [10] où un échantillonnage est effectué sur le modèle probabiliste pour estimer le rang d'un mot donné en entrée. Pour cela, on génère aléatoirement un ensemble W de n mots en respectant les distributions de probabilités du modèle probabiliste. Le tirage d'un mot w_j se fait de cette manière : on tire une base composée $\mathcal{Q} = B_1...B_k$ avec probabilité p_S puis on tire pour i de 1 à k un terminal T_i avec probabilité p_{B_i}. On a $w = T_1...T_k$. Cette opération est répétée n fois. Le rang $R(w)$ d'un mot est estimé à une valeur $\tilde{R}(w)$ définie telle que

$$\tilde{R}(w) = \sum_{w_j \in W} \begin{cases} \frac{1}{S(w_j).n} & \text{si } S(w_j) > S(w), \\ 0 & \text{sinon.} \end{cases}$$

Cet échantillonnage peut être pré-calculé, et il peut être utilisé quelque soit le modèle probabiliste car uniquement le tirage aléatoire du mot w_j dépend du modèle.

7.2.2 Mise à jour des paramètres de ZXCVBN

ZXCVBN pourrait gagner en précision si ses paramètres étaient mis à jour durant son utilisation au sein d'un service. On peut par exemple mettre à jour le dictionnaire embarqué afin d'y ajouter les mots de passe qui deviennent trop populaires. Dès que le nombre d'occurrences d'un mot de passe dépasse un seuil prédéfini, le mot de passe est ajouté au dictionnaire.

En plus de mettre à jour le dictionnaire, il devient nécessaire de mettre à jour les modèles probabilistes en fonction des mots de passe ajoutés au service, tel que défini dans la partie 7.1.2.2. En outre, si un échantillonnage via la méthode de Monte-Carlo a été effectué pour approximer le rang d'un mot de passe, il est nécessaire d'effectuer à nouveau cet échantillonnage car les probabilités ont changé. Il peut être intéressant de chercher un mécanisme pour accélérer ce processus de mise à jour de l'échantillon sans avoir à refaire tout l'échantillonnage.

7.2.3 Implications sur la sécurité

Le fait d'intégrer les modèles probabilistes dans la mesure de robustesse nécessite de stocker les modèles entraînés. De ce fait, un attaquant qui arrive à mettre la main dessus a un avantage car il obtient des informations sur la distribution des mots de passe de la base de données. Il peut donc se servir de ce modèle pour construire et exécuter un énumérateur dans le but de casser les mots de passe de la base de données.

Il serait donc intéressant de mettre en place des mécanismes permettant de limiter la quantité d'information qui fuite lorsque le modèle probabiliste entraîné fuite. Dans le papier [7], les auteurs ajoutent du bruit aux fréquences enregistrées des n-grams, et calculent la borne supérieure à la quantité d'information détenue par le système. Ils montrent que cette borne supérieure est petite, ce qui limite grandement la quantité d'information qui fuite. On peut donc s'inspirer de cette méthode pour ajouter du bruit dans le modèle probabiliste entraîné, afin de faire en sorte qu'un attaquant qui se procure ce modèle n'obtienne qu'une quantité petite d'information. Pour cela, on ajoute le bruit sur les distributions de probabilités. Dans le cas de PCFG, on peut remplacer la mise à jour de chaque base composée $\mathcal{Q}_i$ par le calcul

$$p'_S(\mathcal{Q}_i) = \frac{b + q.p_S(\mathcal{Q}_w)}{1 + \Gamma + q}$$

où

$$b = \begin{cases} 1 & \text{si } \mathcal{Q}_i = \mathcal{Q}_w, \\ 1 & \text{avec probabilité } \gamma, \\ 0 & \text{sinon} \end{cases}$$

et où Γ est le nombre de fois où le cas (2), $b = 1$ avec probabilité γ, a été appliqué. On a donc, à chaque mise à jour d'une probabilité pour une base composée qui n'est pas $\mathcal{Q}_w$, un tirage aléatoire pour déterminer si l'on incrémente artificiellement le nombre d'occurrences de $\mathcal{Q}_i$. On peut appliquer le même principe pour les bases

simples, avec éventuellement une probabilité de bruit différente. Les valeurs de γ dans [7] sont de l'ordre de 10^{-6} et 10^{-7}.

7.3 Conclusion

Si l'on résume les caractéristiques souhaitées d'une métrique de robustesse des mots de passe moderne et efficace, nous avons :

— multimodalité : la métrique exploite les avantages de différentes méthodes de mesure de score afin de tirer les avantages de chacune d'entre elles ;

— adaptabilité : la métrique est paramétrable à l'initialisation, c'est-à-dire qu'elle est capable de prendre en compte des paramètres tels que la langue du service, s'entraîner sur une base de données de mots de passe fournie, afin de proposer une mesure pertinente et personnalisée de la robustesse. De plus, elle présente une procédure de mise à jour de ses modèles probabilistes et de ses paramètres afin d'évoluer durant la vie du service pour proposer une mesure pertinente dans le temps ;

et dans un second temps, une propriété non-négligeable mais moins importante que les deux autres :

— utilisabilité : la métrique est simple à appréhender par les utilisateurs du service, c'est-à-dire qu'elle ne demande pas de compétence particulière, et fait en sorte que les utilisateurs puissent choisir leur mot de passe rapidement, c'est-à-dire que les utilisateurs voient leur mot de passe accepté dans les premières tentatives ;

Évidemment, ces propriétés sont souhaitées, il n'est donc pas évident qu'une métrique de robustesse satisfaisant ces 3 propriétés existe. Nous avons également proposé des pistes d'améliorations pour ZXCVBN, afin qu'il puisse utiliser les modèles probabilistes en respectant les objectifs que nous avons définis. Même nous n'avons pris que ZXCVBN comme exemple, les propositions exposées sont bien évidemment utilisables pour d'autres métriques de robustesse multimodales.

Conclusion

Nous avons étudié dans cette thèse les énumérateurs probabilistes lorsqu'ils sont utilisés pour casser des mots de passe. Nous avons dans un premier temps présenté le cadre dans lequel se déroule cette thèse à travers la définition des termes utilisés, la présentation des outils et des familles de modèles probabilistes. Nous avons également discuté des travaux connexes et positionné notre travail par rapport à ces travaux de l'état de l'art. Le manuscrit ainsi que la thèse réalisée sont structurés autour de trois principaux travaux : la mesure de la performance des énumérateurs dans un contexte d'attaque sur les mots de passe, la mesure de l'efficacité des politiques heuristiques contre les attaques utilisant les modèles probabilistes et finalement l'analyse du comportement des énumérateurs probabilistes d'un point de vue algorithmique. Ces trois travaux nous ont permis de discuter des pistes et des propriétés nécessaires à la conception d'une métrique de robustesse moderne, prenant en compte à la fois les attaques les plus naïves mais également les attaques les plus modernes et précises.

Travaux réalisés et contributions

Le premier travail réalisé consiste en la mesure des performances des énumérateurs dans le contexte d'une attaque sur une base de données de mot de passe protégés par une fonction de hachage. Nous avons dans un premier temps proposé une formalisation de la performance qui tient compte de paramètres importants que sont la vitesse d'énumération des candidats, la vitesse de la fonction de hachage et le taux de succès de l'algorithme d'énumération. Puisqu'il n'est pas possible d'effectuer les mesures exactes nécessaires au calcul de cette performance de manière précise en un temps raisonnable, nous avons dû mettre en place des techniques pour estimer ces valeurs. Une fois ces estimations effectuées, nous avons montré que même les attaques les plus naïves telle que la force brute sont efficaces dans certaines conditions, par exemple lorsqu'une fonction de hachage rapide est utilisée. Nous avons également montré que la lenteur d'un énumérateur n'est pas un problème lorsqu'une fonction de hachage lente est utilisée, à condition que son taux de succès compense sa lenteur. Ce travail permet de montrer que si un énumérateur a pour but de casser des mots de passe, alors il doit être évalué avec la bonne méthode, c'est-à-dire en tenant compte également de sa vitesse d'énumération et la vitesse de la fonction de hachage.

Le second travail réalisé évalue l'efficacité des politiques heuristiques de sécurité des mots de passe pour se protéger des attaques utilisant les modèles probabilistes. En filtrant les bases de données de mots de passe avec les politiques heuristiques étudiées, nous avons conduit des expériences afin de montrer à quel point ces politiques ne sont pas efficaces pour se prémunir de ces attaques. En utilisant les outils formels et techniques présentés dans le premier travail, nous avons montré que même une politique utilisant ZXCVBN, mesure de robustesse présentée comme étant la référence, avec un score requis maximal, ne permet pas de se prémunir suffisamment des attaques utilisant les modèles probabilistes. Les résultats affirment le fait qu'il faille concevoir de nouvelles métriques de robustesse des mots de passe qui tiennent compte de ces modèles probabilistes, car si l'on peut espérer une augmentation de l'utilisation des fonctions de hachage lentes dans les services en ligne, on constatera également une augmentation de l'usage de ces modèles pour en attaquer les mots de passe.

Le troisième travail de cette thèse consiste en l'analyse du comportement de deux énumérateurs probabilistes, OMEN et PCFG, d'un point de vue algorithmique. Nous observons l'évolution des structures de données utilisées dans ces algorithmes, ce qui nous permet de mesurer l'évolution du coût de génération des candidats. Ce travail aborde le problème de l'analyse des performances d'un énumérateur d'un autre angle. En effet, dans la littérature tout comme dans les deux premiers travaux, les énumérateurs sont généralement étudiés en modèle boîte noire, c'est-à-dire que seules les entrées et les sorties sont étudiées. Ici, nous regardons l'intérieur des algorithmes, apportant des informations supplémentaires permettant de concevoir des méthodes pour rendre plus coûteux la génération de candidats. Nous montrons par exemple, que si l'on veut que le coût de génération des candidats dans PCFG soit linéaire en le nombre de candidats générés, alors il faut que la taille de la file de priorité augmente de manière exponentielle, ce qui semble difficile à réaliser en pratique.

Enfin, nous rassemblons les conclusions de chaque travail de ce manuscrit pour proposer des pistes pour concevoir des métriques de robustesse modernes, pour lesquelles nous définissons les caractéristiques que nous jugeons pertinentes à posséder. Même s'il ne semble pas si difficile de proposer une métrique possédant les deux premières propriétés essentielles que sont la multimodalité et l'adaptabilité, il est loin d'être évident de trouver une métrique qui soit également utilisable.

Perspectives de travail

Dans ce manuscrit, un accent a particulièrement été mis sur les performances des énumérateurs et la sécurité des métriques de robustesse. Nous présentons maintenant les perspectives de travail qui pourraient faire suite à cette thèse.

ZXCVBN et modèles probabilistes. Comme proposé de manière détaillée dans le chapitre 7, il nous semble pertinent d'ajouter à ZXCVBN un module d'évaluation du score d'un mot w en utilisant les modèles probabilistes. Ceci requiert d'être capable de calculer un rang d'énumération $r(w)$, ou une estimation, du mot w en fonction de son score $S(w)$ lié à la probabilité de sa décomposition en fonction du modèle. On pourra utiliser la méthode de Monte-Carlo comme suggérée et détaillée

dans le chapitre 7. Une fois ceci réalisé, il convient de mesurer l'utilisabilité de la nouvelle métrique en utilisant les outils des chapitres 5 et 6, par exemple en évaluant le taux de rejet des filtres, c'est-à-dire en calculant quelle proportion des bases de données seraient filtrées par une telle métrique, ou par exemple en conduisant une étude sur l'utilisabilité de cette méthode. ZXCVBN ayant pour vocation d'être assez léger, il serait pertinent également d'étudier l'espace requis par ce système et éventuellement de mettre en place des techniques pour le réduire.

Étude algorithmique des énumérateurs probabilistes. Dans le chapitre 6, nous fournissons la première analyse algorithmique et comportementale de deux énumérateurs probabilistes de l'état de l'art. Pour effectuer ces analyses, nous avons fait des choix sur les opérations que nous considérons comme coûteuses dans la génération des candidats. Dans le cas de PCFG, il est assez clair que l'insertion dans la file de priorité est l'opération la plus coûteuse de l'algorithme. Cependant, la manière dont fonctionne OMEN ne permet pas de simplifier aussi bien l'algorithme que celui de PCFG, car OMEN utilise à la fois des vecteurs et des fonctions récursives qui ne sont pas une seule et unique structure durant toute la génération des candidats. Ceci rend d'autant plus complexe son analyse algorithmique et comportementale. Ce travail d'étude en modèle boîte blanche des énumérateurs probabilistes manque dans le domaine, car la plupart des travaux effectués utilisent ces algorithmes en modèle boîte noire, ce qui ne permet pas toujours de mettre en avant les coûts des algorithmes pour générer des candidats. En étudiant plus précisément et plus profondément les structures de données utilisées, on pourrait proposer des améliorations à ces algorithmes au niveau de l'utilisation mémoire, de leur parallélisation, d'éventuels pré-calculs possibles, ...

Bilan général

Les énumérateurs probabilistes peuvent être comparés à la pêche au chalut ou à la dynamite : on vise une zone et on lance le chalut, puis on récolte les poissons capturés. Plus le chalut est petit et plus il y a de poissons dans la zone, plus la pêche est bonne. En revanche, le chalut est plus coûteux à fabriquer s'il est grand et le poisson plus difficile à remonter s'il y en a beaucoup. En réalité, c'est la **densité de poisson** qui nous intéresse. Tout comme la **densité de mots de passe** : ce que l'on souhaite, c'est essayer le minimum de candidats d'un ensemble pour trouver le maximum de mots de passe. Pour cela, on essaye donc de **minimiser la distance** entre chaque candidat, afin de rendre l'énumération la moins coûteuse possible.

Il nous semble en réalité que la mesure de la robustesse des mots de passe peut se faire depuis deux points de vue :

1. on construit des mesures de robustesse indépendamment des attaques. C'est le cas des métrique heuristiques et des entropies. On évalue ensuite l'efficacité de ces robustesses en conduisant des attaques.

2. on considère un ensemble d'attaques possibles sur les mots de passe, et on construit une métrique qui permette de protéger le plus grand nombre d'entre eux. On évalue ces métriques de la même manière que dans le cas théorique.

Premier axe de recherche : formaliser la notion de distance. De ce point de vue, on souhaite définir une distance qui soit utilisée par un énumérateur de telle sorte que pour retrouver un certain pourcentage de la base de données attaquée, la somme des distances entre les mots générés par l'énumérateur soit minimale. Il y a deux challenges ici : trouver une bonne distance et trouver un énumérateur qui minimise la somme des distances entre chaque couple de mots successifs. Les modèles probabilistes étudiés dans cette thèse permettent déjà de définir une distance entre les mots, correspondant aux différences entre leurs probabilités. Ainsi, pour une distance d et un énumérateur E donnés, il devient possible de comparer la robustesse de deux bases de données D_1, D_2 entre elles. Si on fixe un pourcentage t de la base de données à retrouver, alors on peut les ordonner en fonction de la somme des distances des mots permettant de trouver $t\%$ de ces bases. Même si d n'est pas nécessairement optimale, c'est-à-dire qu'elle permet de minimiser la somme des distances pour tous les énumérateurs, il est possible en revanche qu'elle tienne compte du plus grand nombre d'énumérateurs possible.

Deuxième axe de recherche : construire des mesures en fonction des menaces. C'est le point de vue adopté par les métriques multimodales, car on énumère les différentes attaques possibles et on construit une métrique agrégeant les modalités permettant de se protéger contre chaque attaque. Ici, chaque attaque utilise une distance différente entre les mots, et la mesure de robustesse cherche à espacer les mots le plus possible pour toutes les distances utilisées. Par exemple, l'énumérateur basé sur la grammaire probabiliste de Weir définit la distance entre deux mots proportionnellement à la différence de leurs probabilités : plus deux mots ont une probabilité proche (ce qui est le cas de mots partageant la même base composée), plus la distance les séparant est petite. D'un point de vue implémentation, plus deux mots sont proches, moins il est coûteux de les énumérer à la suite (en termes de nombres d'opérations par exemple).

De part les travaux effectués durant cette thèse, on est amenés à se poser plusieurs questions plus philosophiques sur le domaine.

Premièrement, est-il possible de définir une mesure de robustesse qui soit très satisfaisante, c'est-à-dire une mesure qui soit efficace contre toutes les attaques, même celles qui n'existent pas encore ? On pourrait penser que non, mais rien indique aujourd'hui que c'est impossible. Dans le cas où ce ne serait pas possible, alors la seule solution est de construire ces mesures de robustesse de manière pratique, en se défendant contre chaque attaque et en imaginant des mesures de robustesse multimodales.

Deuxièmement, est-il possible de trouver une distance unique qui soit satisfaisante peu importe l'attaque ? Même si notre intuition nous laisse penser que la réponse est négative, formaliser cette notion de distance est un travail qui peut faire l'objet d'une thèse, et qui apporterait des outils intéressants pour le domaine. Même s'il n'est pas possible de trouver une telle distance, il est peut-être possible d'en trouver une qui soit plus générale et qui regroupe les distances définies par plusieurs modèles probabilistes. On pourrait donc mieux se protéger des plusieurs attaques basées sur ces modèles probabilistes sans avoir à concevoir une métrique multimodale composée de modules protégeant chacun contre une attaque.

Publications de l'auteur

Conférences internationales avec comité de lecture et avec actes

1. **Valois, M.**, Lacharme, P., Le Bars, J-M. Performance of Password Guessing Enumerators Under Cracking Conditions. In 34th International Conference on ICT Systems Security and Privacy Protection (IFIP SEC 2019) (pp. 67-80). Lisbon, Portugal, June 25-27, 2019. Springer International Publishing.
2. Cuan, B., Damien, A., Delaplace, C., **Valois, M.** Malware Detection in PDF Files Using Machine Learning. In 15th International Conference on Security and Cryptography (SECRYPT 2018) (pp. 412-419). Porto, Portugal, July 26-28, 2018. Scitepress Digital Library.

Présentations et Séminaires

1. **Valois, M.**, Lacharme, P., Le Bars, J-M. En quoi les politiques de création de mots de passe peuvent les affaiblir ? Invité aux JRSSI (Journées réunissant les Responsables de la Sécurité des Systèmes d'Information), Novembre 2018, Muséum d'Histoire Naturelle, Paris.

Bibliographie

[1] Jean-Philippe Aumasson. Password hashing competition. `https://password-hashing.net/`, 2015.

[2] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 : new generation of memory-hard functions for password hashing and other applications. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 292–302. IEEE, 2016.

[3] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[4] Joseph Bonneau. The science of guessing : analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy*, pages 538–552. IEEE, 2012.

[5] Joseph Bonneau. Statistical metrics for individual password strength (transcript of discussion). In *International Workshop on Security Protocols*, pages 87–95. Springer, 2012.

[6] William Burr, Donna Dodson, and W Polk. Electronic authentication guideline. Technical report, National Institute of Standards and Technology, 2004.

[7] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from Markov models. In *NDSS*, 2012.

[8] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[9] Dashlane. Kanye west tops dashlane's list of 2018's "worst password offenders". `https://blog.dashlane.com/password-offenders-2018/`, 2018.

[10] Matteo Dell'Amico and Maurizio Filippone. Monte carlo strength evaluation : Fast and reliable password checking. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 158–169. ACM, 2015.

[11] Markus Dürmuth, Fabian Angelstorf, Claude Castelluccia, Daniele Perito, and Abdelberi Chaabane. OMEN : Faster password guessing using an Ordered Markov ENumerator. In *International Symposium on Engineering Secure Software and Systems*, pages 119–132. Springer, 2015.

[12] EFF. "EFF DES cracker" machine brings honesty to crypto debate. `https://web.archive.org/web/20100101001853/http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_descracker_pressrel.html`, 1998.

[13] Tobias Engel. Exploit SS7 to redirect phone calls/sms. `https://attack.mitre.org/techniques/T1449/`, 2014.

[14] Ben Falconers. Adobe crossword. `https://zed0.co.uk/crossword/`, 2013.

[15] Javier Galbally, Iwen Coisel, and Ignacio Sanchez. A new multimodal approach for password strength estimation—part I : Theory and algorithms. *IEEE Transactions on Information Forensics and Security*, 12(12) :2829–2844, 2017.

[16] Javier Galbally, Iwen Coisel, and Ignacio Sanchez. A new multimodal approach for password strength estimation—part II : Experimental evaluation. *IEEE Transactions on Information Forensics and Security*, 12(12) :2845–2860, 2017.

[17] Maximilian Golla and Markus Dürmuth. On the accuracy of password strength meters. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1567–1582. ACM, 2018.

[18] Dan Goodin. Anatomy of a hack : How crackers ransack passwords like "qeadzcwrsfxv1331", 2013.

[19] Jeremie Gosney. 8x Nvidia GTX 1080 Hashcat benchmarks. `https://gist.github.com/epixoip/ace60d09981be09544fdd35005051505`, 2018.

[20] Martin Hellman. A cryptanalytic time-memory trade-off. *IEEE transactions on Information Theory*, 26(4) :401–406, 1980.

[21] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan : A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security*, pages 217–237. Springer, 2017.

[22] Troy Hunt. Have I been pwned, 2017.

[23] Troy Hunt. Here's why [insert thing here] is not a password killer. `https://www.troyhunt.com/heres-why-insert-thing-here-is-not-a-password-killer/`, 2018.

[24] Anil K Jain, Patrick Flynn, and Arun A Ross. *Handbook of biometrics*. Springer Science & Business Media, 2007.

[25] John The Ripper Community. John the ripper implementation. `https://github.com/magnumripper/JohnTheRipper`, 2019.

[26] Peter Kacherginsky. Password analysis and cracking kit. `https://github.com/iphelix/pack`, 2014.

[27] Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Julio Lopez. Guess again (and again and again) : Measuring password strength by simulating password-cracking algorithms. In *2012 IEEE Symposium on Security and Privacy*, pages 523–537. IEEE, 2012.

[28] Daniel V. Klein. "foiling the cracker" : A survey of and improvements to password security. In *Usenix Security Workshop*, 1992.

[29] Saranga Komanduri. *Modeling The Adversary To Evaluate Password Strength With Limited Samples*. PhD thesis, Microsoft Research, 2016.

[30] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people : measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2595–2604. ACM, 2011.

[31] KoreLogic. LibPathWell. `https://github.com/KoreLogicSecurity/libpathwell`, 2015.

[32] RSA Laboratories. `https://tools.ietf.org/html/rfc2898#section-5.2`.

[33] T. Alexander Lystad. Passpal, password analysis software. `https://github.com/arex1337/Passpal`, 2012.

[34] James L Massey. Guessing and entropy. In *Information Theory, 1994. Proceedings., 1994 IEEE International Symposium on*, page 204. IEEE, 1994.

[35] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean, and accurate : Modeling password guessability using neural networks. In *USENIX Security Symposium*, pages 175–191, 2016.

[36] Robert Morris and Ken Thompson. Password security : A case history. In *Communications of the ACM*, volume 22, pages 594–597. ACM, 1979.

[37] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM conference on Computer and communications security (CCS)*, pages 364–372. ACM, 2005.

[38] NIST. Nist special publication 800-63b. `https://pages.nist.gov/800-63-3/`, 2017.

[39] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference (CRYPTO)*, pages 617–630. Springer, 2003.

[40] OpenWall. John the ripper. `http://www.openwall.com/john`, 2017.

[41] Colin Percival. Stronger key derivation via sequential memory-hard functions. *BSDCan*, pages 1–16, 2009.

[42] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. Technical report, IETF, 2016.

[43] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[44] s3in !c. Hashes.org. `https://hashes.org`, 2017.

[45] Sean M. Segreti. Diversify to survive : Making passwords stronger with adaptive policies. In *USENIX Symposium on Usable Privacy and Security*. USENIX, 2017.

[46] Jens Steube. Probability infinite chained elements. `https://hashcat.net/events/p14-trondheim/prince-attack.pdf`, 2014.

[47] Jens Steube. Hashcat implementation. `https://github.com/hashcat/hashcat`, 2019.

[48] Blase Ur, Sean M Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Saranga Komanduri, Darya Kurilova, Michelle L Mazurek, William Melicher, and Richard Shay. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 463–481, 2015.

[49] Mathieu Valois, Patrick Lacharme, Jean-Baptiste Jorand, and Yannick Bass. C++ implementation of PACK with additionnal features. `https://git.unicaen.fr/passwords/cppack`.

[50] Chun Wang, Steve T. K. Jan, Hang Hu, Douglas Bossart, and Gang Wang. The next domino to fall : Empirical analysis of user passwords across online services. In *Proc. of CODASPY*, 2018.

[51] Ding Wang, Haibo Cheng, Ping Wang, Xinyi Huang, and Gaopeng Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11) :2776–2791, 2017.

[52] Rick Wash, Emilee Rader, Ruthie Berman, and Zac Wellmer. Understanding password choices : How frequently entered passwords are re-used across websites. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*, pages 175–188, 2016.

[53] Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 162–175. ACM, 2010.

[54] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE Symposium on Security and Privacy*, pages 391–405. IEEE, 2009.

[55] Dan Lowe Wheeler. ZXCVBN : Low-budget password strength estimation. In *Proc. USENIX Security*, 2016.

[56] Robin Wood. Pipal, password analyser. `https://digi.ninja/projects/pipal.php`.

Annexe A

Version C++ de PACK

Résumé

Dans cet annexe, nous présentons un outil permettant d'analyser une base de mots de passe en calculant des statistiques sur ces derniers. Les principaux inconvénients des solutions actuelles sont leur lenteur, leur abandon et les problèmes d'encodage des caractères. Par le biais de ce projet étudiant, nous proposons une solution rapide, parallélisable et peu coûteuse en mémoire pour répondre à ces inconvénients. Cet outil a pour but d'être intégré facilement dans la chaîne de l'analyse de sécurité des mots de passe, que ce soit pour les tests d'intrusion, les chercheurs en sécurité ou bien les casseurs de mots de passe. Même s'il est orienté vers son usage en ligne de commande, il possède également un mode graphique qui offre une approche plus visuelle du problème afin de sensibiliser un plus large public aux problématiques liées à la robustesse des mots de passe.

A.1 Introduction

La sécurité des mots de passe repose beaucoup sur les informations qu'un attaquant possède sur ceux-ci à priori : sa facilité à les attaquer grandit avec la quantité d'informations qu'il accumule concernant ces mots. C'est pourquoi un attaquant qui souhaite casser des mots de passe effectue généralement une étape d'analyse statistique des mots de passe déjà cassés pour y extraire des motifs récurrents.

Divers outils open-source existent pour effectuer de telles analyses sur des bases de données de mots de passe : PACK [26], Passpal [33], Pipal [56], le premier étant la référence dans la communauté. Cependant, étant tous écrits avec des langages interprétés, leur temps d'exécution peut s'avérer très long. Par exemple, sur une base contenant environ 500 millions de mots de passe, pour une taille de près de 5Go, PACK met 45 minutes pour terminer. Passpal et Pipal ne sont pas plus rapide. De plus, aucun de ces outils ne permet la parallélisation, alors qu'il semblerait que le calcul de ces statistiques puisse en profiter grandement. Aussi, ces outils ne sont plus maintenus et utilisent des versions obsolètes des langages, ce qui pose des problèmes, par exemple l'encodage des caractères pour Python 2 entre autres.

Nous présentons ici l'outil ainsi que les différents challenges techniques qu'il a fallu résoudre. De plus, on étudie l'apport du parallélisme dans ce processus d'analyse et les conséquences sur l'usage de la mémoire vive. Nous répondons ici à ces questions.

A.2 Bref Historique

Ce projet est à l'origine un projet annuel étudiant dans le cadre du diplôme d'ingénieur de l'ENSICAEN. Il a été réalisé par Jean-Baptiste Jorand et Yannick Bass sous licence MIT. Nous avons ensuite repris le projet afin d'ajouter des fonctionnalités présentes dans PACK. Ce projet s'inscrit dans le cadre de la thèse intitulée "Mesure de la Robustesse des Mots de Passe", réalisée par Mathieu et encadrée par Patrick.

A.3 L'outil

Le code source de l'outil, disponible sur notre dépôt git [49], est écrit en C++ pour des raisons de performances. Une fois compilé, il se présente sous la forme d'un exécutable à utiliser en ligne de commande. Il prend en argument le chemin vers le fichier contenant la liste de mots de passe à analyser (un mot par ligne), ainsi qu'une liste d'options et de paramètres. Il calcule ensuite différentes statistiques sur ces mots en nombre et proportion en fonction de :

— leur taille

— leur jeu de caractères

— leur masque simple

— leur masque avancé

Par exemple, le mot de passe "P@ssword123" est de taille 11, utilise le jeu de caractères "minuscule, majuscule, chiffre, caractère spécial" (lowercase, uppercase, digit, special), a pour masque simple "majuscule, caractère spécial, minuscule, chiffres" (USLD), et pour masque avancé "majuscule, caractère spécial, minuscule, minuscule, minuscule, minuscule, minuscule, minuscule, chiffre, chiffre, chiffre" (USLLLLLLDDD ou USL_6D_3). Ces statistiques permettent de déceler des similarités entre les mots de passe, mais aussi d'en apprendre sur les politiques de création des mots de passe : s'il n'existe pas de mots plus court que 6 caractères, c'est très probablement qu'ils ont été interdits. La figure A.1 montre un exemple de sortie de l'outil.

A.3.1 Filtres

L'outil permet également de filtrer les mots à analyser grâce à des expressions régulières afin de rejetter des mots qui seraient aberrants ou ayant une trop grosse influence sur les statistiques. Par exemple, dans la base de données Linkedin, beaucoup de mots de passe de taille 15 n'utilisant que des chiffres sont présents, on peut donc soupçonner un comportement automatique tel un robot. Ces mots n'ont donc pas beaucoup d'intérêt dans notre analyse.

```
Selected 496672991 on 496672991 passwords
(100 %)

Security rules :
Minimal length of a password: 8
Minimum of special characters in a password: 0
Minimum of digits in a password: 1
Minimum of upper characters in a password: 1

--> 4966732 passwords  (1 %) respect the security rules

min - max:
digit:   0 - 470
lower:   0 - 318
upper:   0 - 150
special:   0 - 133

Statistics relative to length:
8:  31,69%    (157416701)
10:  16,56%    (82298172)
9:  13,53%    (67249475)
7:  9,615%    (47757564)

Statistics relative to charsets:
loweralphanum:  45,91%    (228049679)
loweralpha:  24,36%    (120999045)
numeric:  8,605%    (42741330)
mixedalphanum:  6,799%    (33772341)

Statistics relative to simplemasks:
lowerdigit:  30,47%    (151370975)
lower:  24,36%    (120999045)
digit:  8,605%    (42741330)
digitlower:  6,013%    (29868932)
lowerdigitlower:  3,767%    (18712738)

Statistics relative to advancedmask:
?l?l?l?l?l?l?l?l?l:  15,06%    (74847808)
othermasks:  11,68%    (58034271)
?l?l?l?l?l?l?l?l?l:  2,143%    (10644579)
```

FIGURE A.1 – Un exemple de sortie de l'outil

Pour calculer les proportions des masques simples et avancés, une table d'association est utilisée. Dans le cas où il existe beaucoup de masques différents, cette table grossit vite et le programme passe son temps à résoudre les collisions de clés. C'est pourquoi une option est disponible pour regrouper les masques plus long qu'une taille donnée, car en pratique ils sont très peu fréquents. Cela fait aussi gagner de l'espace mémoire.

A.3.2 Parallélisme

Une fonctionnalité majeure demandée était l'utilisation du parallélisme afin de gagner en performance, un défaut important des anciens outils. Pour cela l'outil construit des *threads POSIX* de C++, et chaque thread se charge de traiter une partition du fichier. Le thread principal va lire une première fois le fichier afin de compter le nombre de lignes, puis il donne à chaque thread le numéro de la ligne du début et de fin de sa partition. Ainsi, chaque thread va lire sur le disque la partie du fichier qui lui correspond et effectuer le traitement de ses lignes. Une fois que les threads ont terminé leur traitement, le thread principal s'occupe de fusionner les statistiques effectuées sur les partitions De cette manière, aucune mémoire n'est utilisée pour stocker les mots du fichier, et c'est avantageux car les fichiers sont souvent très gros. Ceci nécessite par contre de lire une première fois le fichier pour compter le nombre de ligne.

Cependant, cette manière de procéder ne peut pas fonctionner lorsque le fichier n'est pas remontable, par exemple avec l'entrée standard, puisque nous avons besoin de lire deux fois le fichier. La solution provisoire trouvée consiste à stocker la liste des mots en mémoire puis de répartir le travail, comme pour un fichier sur le disque. Cela a l'inconvénient de consommer beaucoup plus d'espace mémoire, nous recommandons ce mode uniquement si vous avez suffisamment de mémoire vive et que vous ne pouvez pas stocker le fichier sur le disque. Une manière plus élégante et fonctionnelle serait d'utiliser une thread pool : les threads sont tous créés dès le début et des petites tâches leur sont données régulièrement. De cette manière, il n'y a pas besoin de connaître le nombre de lignes du fichier pour procéder à l'analyse, et le fonctionnement serait le même avec un fichier sur le disque qu'avec un flux de mots depuis l'entrée standard. Néanmoins, il ne semble pas y avoir de manière standard en C++ d'utiliser une thread pool, c'est pourquoi nous réfléchissons actuellement à une solution qui nous convienne.

A.3.3 Gestion de l'UTF-8

Un défaut majeur des anciens outils est leur manque de gestion des caractères unicode. En français par exemple, les caractères accentués étaient ignorés ce qui faussaient les résultats. Notre outil supporte désormais les caractères UTF-8 à condition que le fichier le soit aussi.

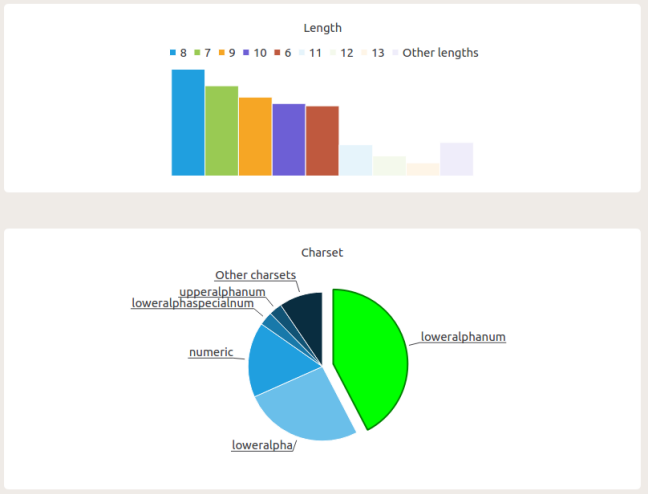


FIGURE A.2 – Les graphiques de l'interface visuelle

A.3.4 Règles de Sécurité

Un aspect intéressant dans l'analyse d'une liste de mots de passe est l'adéquation ou non des mots à différentes règles de composition. Par exemple, combien de mots satisfont les conditions "plus long que 8", "avec au moins 1 caractère spécial", "possède au moins deux chiffres" ? Cela permet par exemple, dans le contexte d'une analyse de sécurité d'une entreprise, d'avoir un aperçu de combien de mots devront être modifiés lors d'un changement de politique de création de mots de passe de l'entreprise.

A.3.5 Interface Graphique

Nous avons aussi réalisé une interface graphique à cet outil afin de pouvoir le présenter lors d'évènements de médiation scientifique tels que la fête des sciences. L'outil reste principalement orienté ligne de commande, l'usage de l'interface graphique n'est donc qu'optionnel. L'interface, où les résultats sous forme de graphiques sont visibles sur la figure A.2, permet de charger le fichier des mots de passe et de sélectionner les options comme en ligne de commande, et affiche un histogramme et un camembert sur les statistiques globales. Cette interface est réalisée en Qt5 en requiert la librairie Qtcharts pour la partie graphique. La version 18.04 permet d'installer facilement les dépendances avec les paquets *qt5-default* et *libqt5charts5-dev*.

A.4 Résultats

Sur la base de 500 millions de mots, PACK mettait 45 minutes pour terminer. Ici, simplement en changeant de langage de programmation, nous mettons 7 minutes pour terminer sur le même fichier, soit 6.5 fois moins que PACK, sans parallélisme.

La parallélisation de l'analyse s'avère efficace, jusqu'à un certain point. La figure A.3 nous montre l'évolution du temps de calcul en fonction du nombre de

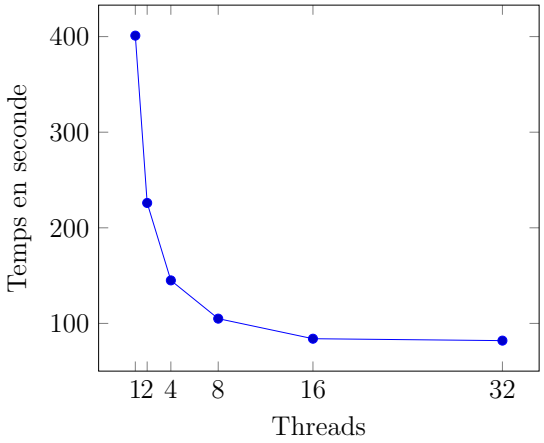


FIGURE A.3 – Temps de calcul en seconde en fonction du nombre de threads utilisés

threads utilisés. Nous constatons que jusqu'à 8 threads, l'apport est important, d'un facteur 4 entre 1 et 8 threads, réduisant ainsi le temps de calcul à environ 1,5 minutes. Au delà de 8 threads, il n'y a plus de gain, car chaque thread n'a plus assez de travail pour compenser la lecture du fichier et la fusion des résultats par le thread principal.

A.5 Améliorations Futures

Une fonctionnalité intéressante et qui pourrait faire le lien avec la recherche sur le thème de la mesure de la robustesse des mots de passe est la détection d'utilisation de règles de réécriture, à la manière du script "rulegen" de PACK. Par exemple, connaître le nombre de mots de passe qui dérivent du mot "password" ("p@ssword", "password123", "P@ssword123!").

En utilisant une threads pool, il est possible que le temps d'exécution soit encore amélioré, car la gestion des threads par le langage est sûrement plus efficace que de le faire à la main. Cela permettrait aussi de pouvoir utiliser correctement l'outil avec l'entrée standard, ce qui permettrait d'utiliser l'outil dans une chaîne de commande afin d'éviter de stocker de trop gros fichiers contenant les mots de passe.

A.6 Conclusion

La solution proposée par le biais de ce projet étudiant a pour vocation de remplacer PACK de manière transparente car c'est un outil déjà très utilisé dans la communauté des chercheurs en sécurité des mots de passe. C'est pourquoi l'affichage des résultats est très proche de celui de PACK, et que les différentes options pour traiter les résultats ultérieurement sont aussi disponibles. Il est par exemple possible d'écrire les masques avancés calculés dans un fichier pour s'en servir avec les logiciels de cassage de mots de passe Hashcat [47] et John The Ripper [40].

L'interface graphique est aussi un moyen visuel de montrer au grand public à quel point la sécurité de nos mots de passe est faible. Comme dans la capture présentée en figure A.2, on peut pointer la trop grosse partie des mots de passe qui ne sont composés que de minuscules, que de chiffres ou qui sont assez courts.

Table des figures

À l'ère où notre identité numérique se confond toujours davantage avec notre identité personnelle, les besoins en sécurité de nos comptes en ligne sont d'autant plus marqués. Les mots de passe sont à la fois la manière de s'authentifier la plus utilisée et à la fois le maillon le plus faible de la chaîne de sécurité. Malgré l'indéniable fragilité de la plupart des mots de passe utilisés en ligne, le mot de passe reste le meilleur moyen de s'authentifier réunissant sécurité, accessibilité et respect de la vie privée.

L'objectif de cette thèse est de faciliter la conception de mesures de robustesse des mots de passe qui soient pertinentes vis-à-vis des attaques les plus sophistiquées sur les mots de passe. Ces attaques reposent sur des modèles probabilistes de la manière dont les utilisateurs choisissent leur mot de passe. Il s'avèrent que ces attaques sont très efficaces pour trouver des mots de passe plus complexes que ceux habituellement trouvés par les méthodes naïves. Ce travail repose sur trois contributions pour identifier les points clés d'une mesure de robustesse des mots de passe moderne. La première contribution modélise le processus d'attaque sur les mots de passe, formalise et mesure la performance d'un tel processus. La deuxième contribution se charge de montrer que les méthodes actuellement déployées pour mesurer la robustesse des mots de passe ne sont pas suffisantes pour se prémunir des attaques sophistiquées. La troisième contribution analyse algorithmiquement les attaques sophistiquées en observant leur comportement dans le but de concevoir des méthodes efficaces pour augmenter le coût d'exécution de ces attaques. La validation des méthodes s'est effectuée en utilisant des mots de passe issus de fuites de données publiques, totalisant plus de 500 millions de mots de passe.

Password Strength Measurement

At the era where our digital identity is always more linked with our personal identity, the security needs of our online accounts become more significant. Passwords are both the most used authentication mean and the weakest link in the security chain. Despite the undeniable weakness of passwords, they remain the best authentication factor which gathers security, accessibility and privacy protection.

The purpose of this thesis is to ease the design of passwords strength measurement methods. Such methods needs to be relevant according the most sophisticated attacks on passwords. Such attacks lay on probabilistic models which model the way that passwords are chosen. These attacks are very efficient to find more complex passwords that are usually not found by naive techniques. To identify the requirements for a modern password strength measure, the work is driven by three contributions. The first contribution shows how we model the attack process on passwords, formalize and measure the performance of such a process. The second contribution shows that currently deployed strength measurement techniques lack to protect passwords against sophisticated attacks. In the third contribution, we analyse the algorithms of sophisticated attacks by observing their behaviour, we the aim of designing techniques that increase their execution cost. The methods has been validated by using passwords from publicly disclosed leaks, for a total of more than 500 millions passwords.

Spécialité Informatique. Mots-clés : Sécurité, mot de passe, robustesse, modèles probabilistes, énumérateurs, fonctions de hachage

ENSICAEN - UNICAEN - CNRS - GREYC UMR 6072, F-14050 Caen, France