

Rapport

OUTIL D'ANALYSE STATISTIQUE

Février 2018

Jean-Baptiste JORAND
Yannick BASS

Tuteurs / Clients : - Patrick LACHARME
- Mathieu VALOIS



REMERCIEMENTS

Nous tenons à remercier Patrick LACHARME et Mathieu VALOIS pour leur aide et leurs conseils tous au long de ce projet. Nous remercions également le GREYC de nous avoir accueilli dans leur locaux pour la durée de ce projet.

TABLE DES MATIERES

1. Introduction	5
2. Etat de l'art	6
2.1. Les outils existants	6
2.2. Les formats de stockage des mots de passe	7
3. Présentation de l'outil	8
3.1. Langage de programmation	8
3.2. Encodage des fichiers et caractères spéciaux	8
3.3. Statistiques calculées	9
3.4. Structures	11
3.4.1. Classification	11
3.4.2. Définition des types	11
3.4.3. Conteneur	12
3.5. Expressions régulières	12
3.6. Programmation multi-cœur	12
3.7. Options d'optimisation	14
3.8. Interface graphique	15
4. Méthode de travail	16
5. Planning du projet	17
6. Bilan du projet	18
7. Bilan personnel des étudiants	18

TABLE DES FIGURES

Figure 1 : format "classique", format "withcount"	7
Figure 2 : exemple d'affichage sur console	10
Figure 3 : Evolution du temps en fonction du nombre de threads sur rockyou	13
Figure 4 : Evolution du temps sur un rockyou concaténé 10 fois	14
Figure 5 : Interface graphique	15

TABLE DES TABLEAUX

Tableau 1 : avantages et inconvénients de chaque outil	6
Tableau 2 : temps d'exécution des outils en fonction de la taille de la base de données	6
Tableau 3 : Statistiques pertinentes sur les mots de passe	9
Tableau 4 : influence des maps sur le temps d'exécution	11
Tableau 5 : Structures créées	12
Tableau 6 : Planning du projet	17

PRESENTATION DU PROJET

Présentation du projet et de ses objectifs.

1. Introduction

De nombreux serveurs créent leur propre base de données pour enregistrer leurs utilisateurs via leur login et leur mot de passe. Néanmoins, celle-ci peut être attaquée et les mots de passe des utilisateurs divulgués sur internet. Cela a été par exemple le cas du site « *rockyou* » en décembre 2009. Les mots de passe y étaient stockés en clair.

Une fois divulgués, ces derniers ont été utilisés par de nombreuses études statistiques. Le but de celles-ci était, entre autre, de déterminer la structure des mots de passe utilisés par les utilisateurs, i.e. la taille, le nombre de caractères minuscules, majuscules, numériques et spéciaux, aussi appelé masque *hashcat*. Des outils dédiés à cette analyse ont été créés par la communauté scientifique.

Toutefois, le temps d'exécution de ces outils est conséquent pour une base de données importante. En effet, celle de « *rockyou* » contient par exemple plus de 32 millions de mots de passe, pour une taille de 290 megas octets, et son analyse peut durer plus d'une heure selon les outils. Pourtant, celle-ci est considérée comme une « petite » base de données, les plus grandes pouvant atteindre plusieurs gigas octets.

Ainsi, l'utilisation de ces outils n'est plus viable pour une base de données plus développée. Nos tuteurs et clients, Patrick Lacharme et Mathieu Valois, nous ont donc posé la problématique suivante : comment pourrait-on donc optimiser les méthodes d'analyse afin d'observer ces statistiques avec un faible temps d'exécution, sans pour autant être excessivement coûteux en mémoire ?

Nous avons ainsi créé notre propre outil d'analyse, répondant au problème posé précédemment.

2. Etat de l'art

2.1. Les outils existants

Trois principaux outils d'analyse statistique de mots de passe sont disponibles en Open Source, à savoir Pack, Pipal et Passpal (lien vers les sources dans la *webographie*). Les tests sur Passpal n'ont pas pu être réalisés suite à une incompatibilité entre le code disponible et les dernières versions de Ruby.

Ces trois outils s'appuient sur des langages interprétés tels que Python et Ruby. Le tableau 1 représente un comparatif entre ces deux outils

	PACK	PIPAL
AVANTAGES	- Gestion des masques <i>hashcat</i> .	- De nombreuses statistiques disponibles ; - Graphiques statistiques disponibles.
INCONVENIENTS	- Impossible de sélectionner uniquement les statistiques qui nous intéressent.	- Caractères spéciaux non traités ; - Les espaces dans les mots de passe ne sont pas pris en compte.

Tableau 1 : avantages et inconvénients de chaque outil

Les résultats obtenus dans le tableau 2 montre que Pack est bien plus performant que Pipal. Il était donc plus intéressant de s'appuyer sur le code de Pack pour développer un nouvel outil ne s'appuyant pas sur un langage interprété.

TAILLE DE LA BASE DE DONNEES	PACK	PIPAL
159.9 Mo	1min 28sec	41min
290.0 Mo	3min 17sec	Nan
580.0 Mo	5 min 11 sec	Nan

Tableau 2 : temps d'exécution des outils en fonction de la taille de la base de données

2.2. Les formats de stockage des mots de passe

Il existe deux principaux formats de stockages des mots de passe. Le premier est une simple liste de l'ensemble des mots de passe les uns à la suite des autres à raison d'un mot de passe par ligne. C'est le format le plus simple à lire. Le second format est le *withcount* qui ne contient chaque mot de passe qu'une seule fois, Ceux-ci sont alors précédés par leur nombre d'occurrences afin de garder les mêmes statistiques que le format classique mentionné plus haut pour une même liste de mot de passe. Ce second format n'est pas géré par les outils actuels, mais nous avons souhaité implémenter sa gestion, car il nous paraissait plus rapide d'analyser un fichier au format *withcount* plutôt qu'un fichier au format classique.

123456	290729	123456
12345	79076	12345
123456789	76789	123456789
password	59462	password
iloveyou	49952	iloveyou
princess	33291	princess
1234567	21725	1234567
rockyou	20901	rockyou
12345678	20553	12345678
abc123	16648	abc123
nicole	16227	nicole
daniel	15308	daniel
babygirl	15163	babygirl
monkey	14726	monkey
lovely	14331	lovely
jessica	14103	jessica
654321	13984	654321
michael	13981	michael
ashley	13488	ashley
qwerty	13456	qwerty

Figure 1 : format "classique", format "withcount"

OUTIL D'ANALYSE STATISTIQUE

Présentation de l'outil réalisé au cours de ce projet

3. Présentation de l'outil

3.1. Langage de programmation

Les outils existants ont tous été développés avec des langages interprétés notamment pour un développement plus rapide, (Python a aussi l'avantage de ne pas poser de problèmes avec des encodages incorrects dans les fichiers stockant les mots de passe). Cependant, de par leur manière de fonctionner, les langages interprétés sont plus lents que les langages compilés. Comme nous souhaitons obtenir les meilleures performances possibles nous avons choisi un langage compilé pour notre projet, le C++.

3.2. Encodage des fichiers et caractères spéciaux

Les listes de mots de passe peuvent être encodés de plusieurs manières différentes voire mal encodées. L'outil étant réalisé en C++, il est plus sensible aux problèmes d'encodage que Pack qui est réalisé en Python. L'outil a été conçu pour travailler avec des fichiers encodés en UTF-8. Pour cela, une commande Unix, "iconv", permet de convertir la liste de mots de passe en un fichier codé en UTF-8. Si ceci n'est pas respecté, certains caractères spéciaux peuvent être codés sur plusieurs octets. Ils seront incorrectement lus par notre programme et seront analysés comme une suite de caractères spéciaux, faussant ainsi les statistiques finales. Dans le pire des cas ils seront interprétés comme un caractère de fin de fichier provoquant la fin de l'analyse de manière prématurée.

De plus, pour prendre en compte tous les caractères possibles au format Unicode, nous n'utilisons plus des *strings*, mais des *wstrings*. *Wstring* représente une suite de caractères, utilisant le type *wchar_t* et non le type *char*. Un *wchar_t* a la particularité d'être codé sur quatre octets, et donc peut correspondre à un caractère non-ASCII, comme les caractères chinois par exemple. Notre outil est donc capable d'analyser tous les caractères.

3.3. Statistiques calculées

L'intérêt principal de cet outil est de ne s'intéresser qu'aux statistiques les plus pertinentes sur les mots de passe. Ces statistiques sont générées en analysant chaque lettre du mot de passe une à une et en déterminant à quelle catégorie elle appartient (majuscule, minuscule, chiffre, caractère spécial). À partir d'une liste de mots de passe, le programme affichera donc les probabilités représentées dans le tableau 3 suivant.

STATISTIQUES	EXEMPLES	INTERET
NOMBRE MINIMUM ET MAXIMUM DE CHAQUE TYPE DE CARACTERE	lower : 0 - 20 pas plus de 20 minuscules dans un mot / certains mots sans minuscule	Savoir qu'un mot de passe ne dépassera jamais plus de 20 minuscules dans notre exemple
LONGUEUR DU MOT DE PASSE	6 caractères	Connaître la taille des mots de passe les plus probables
TYPE DU MOT DE PASSE	loweralphanum : minuscule avec chiffre (pas d'ordre indiqué)	Connaître le type de caractères présents dans les mots de passe les plus probables
MASQUES SIMPLES	lowerdigit : suite de minuscules suivis de chiffre	Connaître le format simplifié des mots de passe les plus probables
MASQUES AVANCEES	?l?l?l?d?d : trois minuscules suivis de deux chiffres	Connaître l'allure précise des mots de passe les plus probables
POURCENTAGE DE MOTS DE PASSE ANALYSE	Inférieur à 100% uniquement si une expression régulière est utilisée	Connaître le pourcentage des mots de passe qui respecte l'expression régulière
POURCENTAGE DES MOTS RESPECTANT LES REGLES DE SECURITES	x%	Connaître le taux de mots de passe validant les règles de sécurité

Tableau 3 : Statistiques pertinentes sur les mots de passe

Concernant les règles de sécurité, seules des valeurs par défaut ont été enregistrées, à savoir :

- taille minimale d'un mot de passe requise : 8 ;
- nombre minimum de caractères minuscules requis : 1 ;
- nombre minimum de caractères majuscules requis : 1 ;
- nombre minimum de chiffres requis : 1 ;
- nombre minimum de caractères spéciaux requis : 0.

Néanmoins, celles-ci peuvent être mises à jour selon les préférences de l'utilisateur.

La figure 2 représente un exemple d'affichage de notre outil. Il s'agit de l'analyse de la liste de mots de passe *rockyou* au format *withcount*.

```
Selected 3.2603e+07 on 3.2603e+07 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 lower characters in a password: 1
  Minimum of upper characters in a password: 1

--> 397975 passwords      (1.22067 %) respect the security rules

min - max

          digit:  0 - 255
          lower:  0 - 255
          upper:  0 - 187
        special:  0 - 255

Statistics relative to length:

          6: 26,03%      (8.4884e+06)
          8: 19,97%      (6.5131e+06)
          7: 19,28%      (6.28802e+06)
          9: 12,11%      (3.94984e+06)
         10: 9,062%      (2.95464e+06)

Statistics relative to charsets:

        loweralpha: 41,68%      (1.35892e+07)
        loweralphanum: 33,17%      (1.08147e+07)
          numeric: 15,92%      (5.19299e+06)
        loweralphaspecial: 1,641%      (535339)
        upperalphanum: 1,632%      (532259)
          upperalpha: 1,497%      (488159)
        loweralphaspecialnum: 1,442%      (470187)
        mixedalphanum: 1,421%      (463511)
          mixedalpha: 0,863%      (281427)
            all: 0,177%      (57762)
        mixedalphaspecial: 0,171%      (55801)
          specialnum: 0,161%      (52503)
        upperalphaspecial: 0,093%      (30430)
        upperalphaspecialnum: 0,089%      (29123)
            special: 0,029%      (9708)

Statistics relative to simplemasks:

          lower: 41,68%      (1.35892e+07)
        lowerdigit: 27,68%      (9.02639e+06)
          digit: 15,92%      (5.19299e+06)
        digitlower: 2,533%      (826139)
        lowerdigitlower: 1,619%      (527902)
```

Figure 2 : exemple d'affichage sur console

3.4. Structures

3.4.1. Classification

Chaque caractéristique d'un mot de passe est enregistrée dans des *maps*. Une *map* a la particularité d'associer une clé à une valeur. Nous enregistrons alors quatre *maps* avec pour clé respective la taille du mot de passe, son type (*charset*), son masque simple et son masque avancé. À chaque mot de passe analysé, nous déterminons alors ses caractéristiques et les ajoutons aux *maps*. Si la clé n'existe pas, elle sera créée et sa valeur sera initialisée à 1. Sinon, nous itérons sa valeur.

Néanmoins, la création d'une clé dans une *map* se fait avec un tri et un reclassement de chaque clé. Or, le nombre de clés à créer peut être excessivement élevé en fonction des différences entre mots de passe et de leur nombre, notamment pour les masques simples et avancés. Cela ralentit le temps de calcul et ne peut donc pas être accepté. Pour résoudre ce problème, nous utilisons alors des *unordered_maps* qui ont les mêmes fonctionnalités que des *maps* à l'exception qu'ils ne trient pas à chaque insertion de clé.

Ensuite, une fois ces *unordered_maps* remplies, nous ajoutons chaque paire valeur/clé dans une *multimap*, les triant ainsi rapidement par valeur. L'affichage de ces *multimaps* renverra alors les statistiques les plus intéressantes dans l'ordre.

FONCTIONS	TEMPS AVEC MAP	TEMPS AVEC UNORDERED_MAP
LONGUEUR	9.8%	13.1%
CHARSET	13.1%	18.4%
MASQUES SIMPLES	27.9%	21.1%
MASQUES AVANCEES	34.4%	23.7%
TEMPS D'EXECUTION TOTAL	61 secondes	38 secondes

Tableau 4 : influence des maps sur le temps d'exécution

Le tableau 4 représente l'influence de l'utilisation des *maps* sur le temps d'exécution. Nous constatons directement que le temps d'exécution est bien plus faible lorsque nous utilisons des *unordered_maps*.

3.4.2. Définition des types

Les compteurs de mots inclus dans chaque *unordered_map* étaient initialement des *Integer* afin d'économiser le plus de place possible en mémoire. Néanmoins, cela a généré un problème dans le cas d'une grande liste de mots de passe. Les compteurs dépassaient alors la limite des *Integer*. Pour résoudre ce problème, nous avons remplacé les *Integer* par des *Double*, repoussant ainsi la limite des compteurs.

3.4.3. Conteneur

Plusieurs structures ont été créées afin de simplifier la lecture du code. Celles-ci sont détaillées dans le tableau 5.

STRUCTURES	INTERET
THREAD_DATA	Structures nécessaires afin de récupérer toutes les variables essentielles au bon fonctionnement des threads.
POLICY	Contient le nombre de minuscules, majuscules, chiffres et caractères spéciaux pour un mot.
CONTAINER	Contient le <i>charset</i> , les masques simples et avancés, la taille et la <i>Policy</i> d'un mot.
MINMAX	Sauvegarde les minimums et maximums de minuscules, majuscules, chiffres et caractères spéciaux des mots de passe.
SECURITYRULES	Permet de comparer les données de chaque mot de passe avec les règles de sécurité définies.

Tableau 5 : Structures créées

3.5. Expressions régulières

Afin de permettre de sélectionner seulement certains mots de passe à analyser, nous avons implémenté la possibilité de passer en paramètre une expression régulière et ainsi d'ignorer les mots de passe qui ne lui correspondent pas. Nous avons utilisé une librairie standard pour coder cette option et, dû à l'implémentation proposée par cette dernière, l'utilisation des expressions ralentit énormément le programme.

3.6. Programmation multi-cœur

Une fois les fonctionnalités « *de base* » implémentées, la suite logique était d'optimiser au maximum le code afin d'obtenir les résultats voulus le plus rapidement possible. Pour ce faire, nous avons décidé d'orienter notre programme sur des fonctionnalités multi-cœur. Dans un premier temps nous avons essayé d'implémenter une *threadpool*, c'est-à-dire un groupe de thread à qui on envoie du travail en continu. Cependant nous avons eu des difficultés à l'implémenter et nous avons préféré changer de méthode par manque de temps.

Nous avons ensuite essayé d'implémenter le parallélisme via de la programmation bas-niveau avec *OpenMP* (utilisation de directives *#pragma*). Cependant notre programme s'adaptait au final assez peu à cette solution car il nécessite un accès constant au fichier (car impossible de charger de grosses listes de mots de passe en mémoire) et nécessite de lire les caractères dans un certain ordre (pour réaliser les masques *hashcat* on a besoin de lire les caractères de gauche à droite). Les directives ne permettant pas de contrôler dans quel ordre les threads renvoient les différents caractères, cela rendait certaines parties du code impossible à paralléliser. Ces limitations rendaient l'utilisation d'OpenMP peu optimale et nous n'avons pas obtenu de gain avec cette méthode.

Enfin, nous avons choisi de simplement utiliser des threads afin de gérer complètement la parallélisation nous même. Nous sommes parti de l'idée de donner à chaque thread sa partie du fichier, et de le laisser faire l'ensemble des calculs nécessaires, plutôt que de séparer les calculs pour chaque mot de passe. Comme il est inimaginable de séparer la liste de mots de passe en plusieurs fichiers il fallait trouver un moyen de répartir les mots de passe entre les threads. Pour cela le programme parcourt une première fois le fichier afin de compter le nombre de lignes (et donc de mots de passe). Ce nombre est divisé par le nombre de threads afin d'obtenir les bornes des différents threads. Chacun d'entre eux va ensuite aller à lire dans le fichier les mots de passe lui étant attribué et les analyser. Il suffit enfin au programme de réunir les différentes maps pour obtenir les statistiques complètes de la liste.

Les figures suivantes montrent le gain de temps obtenu en fonction du nombre de thread :



Figure 3 : Evolution du temps en fonction du nombre de threads sur rockyou



Figure 4 : Evolution du temps sur un rockyou concaténé 10 fois

3.7. Options d'optimisation

Avant l'utilisation des *unordered_map*, nous avons constaté que l'ajout de clé dans une *map* était plus important dans le cas des masques simples et avancés. Ceci est dû au fait que ces derniers sont bien plus variés que les longueurs des mots de passe et que les *charsets*. Nous avons donc ajouté une option limitant la taille des masques. Cela n'influe pas les statistiques globales car les masques de taille importante sont peu fréquents et donc peu représentatif.

Dans le cas des *maps*, cela permettait d'accélérer le temps d'exécution car moins de valeurs étaient enregistrées dans celles-ci. Néanmoins, avec l'utilisation des *unordered_maps*, le gain en temps d'exécution est devenu négligeable. Cependant, nous gagnons toujours en espace mémoire. En effet, imposer une limite à 10 pour les masques simples et avancés, nous gagnons entre 10 et 15% d'espace mémoire sur *rockyou*.

3.8. Interface graphique

Le projet ayant été terminé en avance, nous avons décidé en accord avec nos tuteurs de développer une interface graphique simple pour l'application. Nous avons conçu cette interface dans un but de démonstration, les statistiques affichées ont donc été simplifiées afin de les rendre plus claires pour les néophytes. Les statistiques non affichées sont stockées dans un fichier "result.txt". Ainsi on ne perd pas d'informations lors de l'utilisation de la version graphique même si seule une petite partie des informations est affichée par l'interface.

Pour intégrer notre code dans une application Qt nous avons dû l'encapsuler dans un Qthread (les threads spécifiques à Qt) afin d'éviter que l'application ne se bloque pendant le calcul des statistiques. Cela ralentit cependant l'analyse car un thread s'assure que la partie graphique ne se bloque pas.

Le paramétrage des différentes options ne fait pas de simples getters et setters et doit être fait avant de lancer l'analyse. En effet pour éviter tous problèmes nous avons décidé de bloquer les champs de saisie durant l'analyse.

Enfin, pour réaliser l'affichage des différentes statistiques nous avons utilisé différents graphiques. En conséquence, pour utiliser cette version graphique il est nécessaire d'avoir installé QtCharts qui n'est disponible que pour les versions de Qt supérieures à 5 (Nous avons personnellement utilisé Qt 5.9.4).

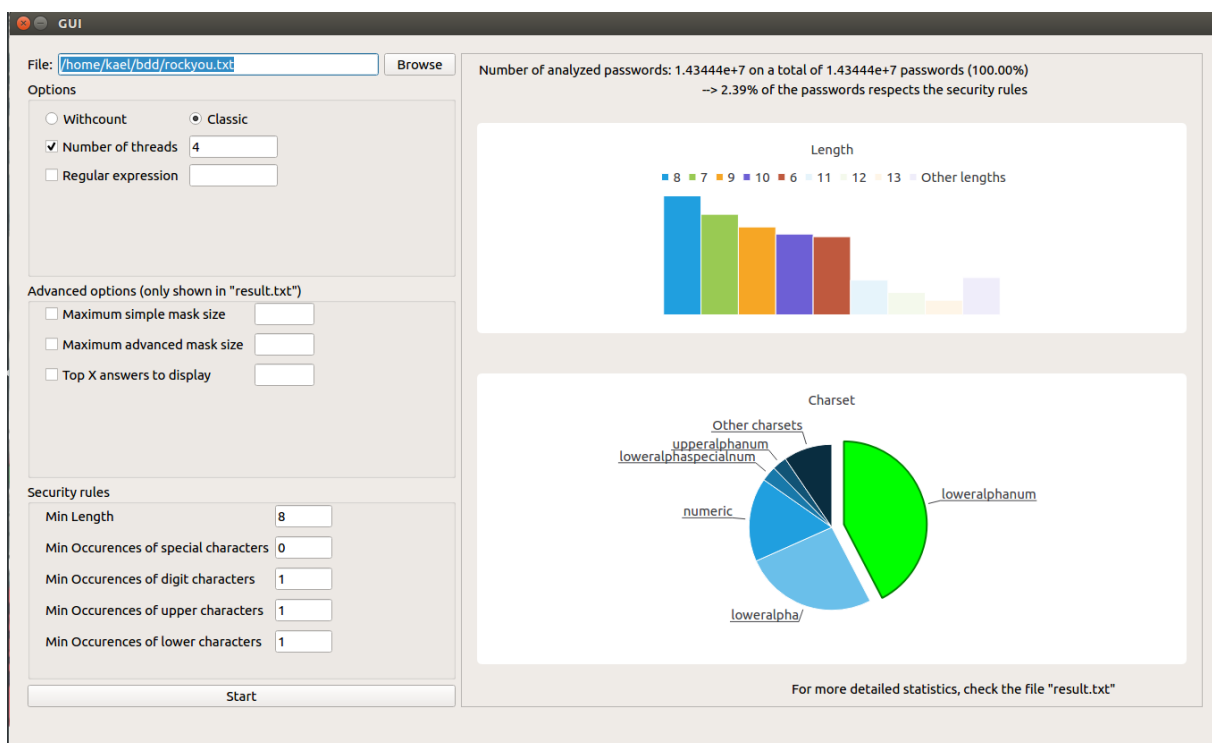


Figure 5 : Interface graphique

GESTION DE PROJET

4. Méthode de travail

Pendant tout le projet nous avons beaucoup communiqué avec nos clients. Nous allions les voir tous les mercredis matins afin de faire un point sur l'avancement du projet et leur présenter les objectifs à réaliser pour la journée. À la fin de ces journées nous leur présentions brièvement le travail effectué ainsi que les éventuelles difficultés rencontrées. De cette manière ils ont pu nous éviter de rester bloquer sur certains problèmes et nous avons pu ajuster notre développement afin de répondre au mieux à leurs attentes. Pour travailler au sein du binôme nous avons utilisé le Gitlab mise à disposition par l'ENSICAEN ainsi qu'un drive Google.

5. Planning du projet

Nous n'avons pas réalisé de planning complet pour prévoir l'évolution de notre projet au cours des séances de travail. À la place, nous connaissons les différents objectifs à atteindre et nous décidions en début de séance quel objectif allait être développé durant la journée, Le tableau suivant présente les différentes tâches que nous avons réalisées à chaque séance.

Date	Objectifs
4 Octobre	Étude de l'existant
11 Octobre	Début du développement
18 Octobre	Gestion de l'encodage des caractères
25 Octobre	Gestion de l'encodage des caractères + Ajout de filtres d'affichage
8 Novembre	Gestion des expressions régulières / Script de transformation <i>withcount</i>
15 Novembre	Gestion du format <i>withcount</i>
22 Novembre	Mesure des temps d'exécution + Ajout d'options d'optimisation
29 Novembre	Soutenance
6 Décembre	Programmation multi-cœur
13 Décembre	Programmation multi-cœur
20 Décembre	Programmation multi-cœur + Documentation
10 Janvier	Gestion des règles de sécurité
24 Janvier	Interface graphique
31 Janvier	Interface graphique
7 Février	Interface graphique + Rapport
14 Février	Rapport + Soutenance
23 Février	Soutenance

Tableau 6 : Planning du projet

6. Bilan du projet

Tous les objectifs du projet ont été remplis et dans le temps restant une interface simple a été créée pour des besoins de démonstration. Notre programme est bien plus rapide que Pack cependant il nécessite du travail en amont afin d'avoir un fichier correctement encodé en UTF-8.

On peut imaginer de nombreux ajouts de fonctionnalités au programme. Par exemple, on pourrait détecter les différentes transformations appliquées à un mot pour qu'il devienne le mot de passe analysé (utilisation de chiffres pour remplacer des lettres en écriture leet par exemple).

Enfin, il est probablement possible d'optimiser encore plus le programme en proposant une meilleure implémentation de la programmation multi-coeur. Notre solution, bien que performante, n'est pas aussi optimal que le serait une threadpool bien implémentée.

7. Bilan personnel des étudiants

Ce projet a été une expérience très enrichissante pour nous. Cela nous a permis de travailler en profondeur sur les mots de passe, qui est un sujet qui nous intéressait tous les deux. Cela nous a également permis d'acquérir plus d'expérience en C++, ce que nous manquions. Enfin cela nous a permis de faire face à des problèmes d'optimisation, problème que nous n'avions jamais eu à nous soucier auparavant et nous a permis de mettre en pratique nos connaissances en programmation parallèle.

CONCLUSION

Bien que nous ayons implémenté toutes les fonctionnalités prévues au début du projet, il reste encore de nombreux ajouts possibles à intégrer au programme. Pour que le programme continue d'évoluer nous avons mis une license libre (license MIT) sur ce projet afin que tous les développeurs ayant un intérêt dans le domaine puissent se baser sur notre travail et continuer de l'améliorer et le partager afin de faciliter la tâche à tous les chercheurs travaillant sur les mots de passe.

Ce projet a été une très bonne expérience pour nous et nous sommes très satisfaits de notre choix de sujet de projet. Nous avons beaucoup appris durant sa réalisation et nous espérons qu'il servira un maximum à nos clients.

WEBOGRAPHIE

- Pack : <https://thesprawl.org/projects/pack>
- Passpal : <http://thepasswordproject.com/passpal>
- Pipal : <https://digi.ninja/projects/pipal.php>



Ecole Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

