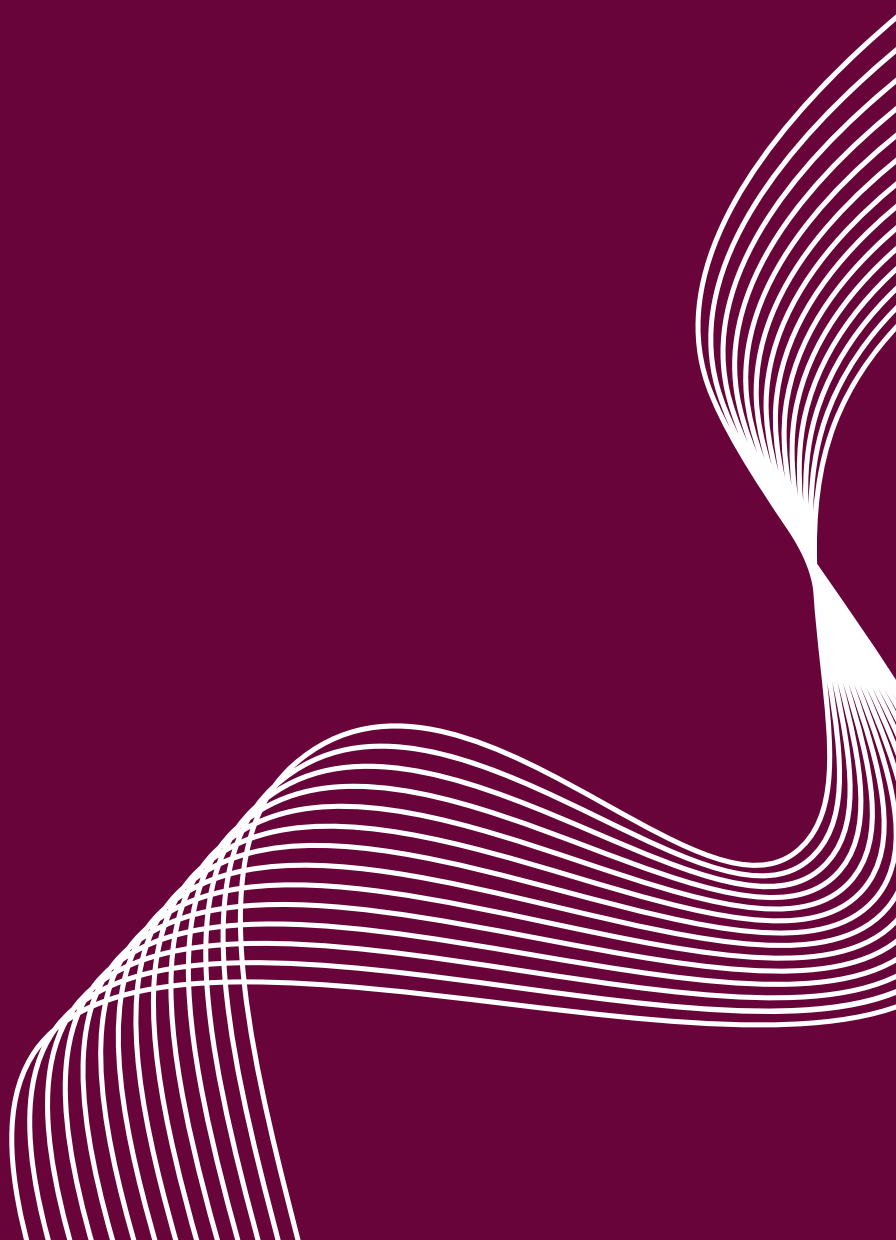
UVSQ université PARIS-SACLAY

UFR des Sciences
CAMPUS DE VERSAILLES

MAI 2023

RAPPORT DE PROJET DE CRYPTOGRAPHIE

Cassage de mot de passe



Préparé par

FRATCZAK Théo
JOLY Thomas
PIRABAKARAN Thanushan
SANTINI Maya

Plan

I . Présentation générale

1. Introduction

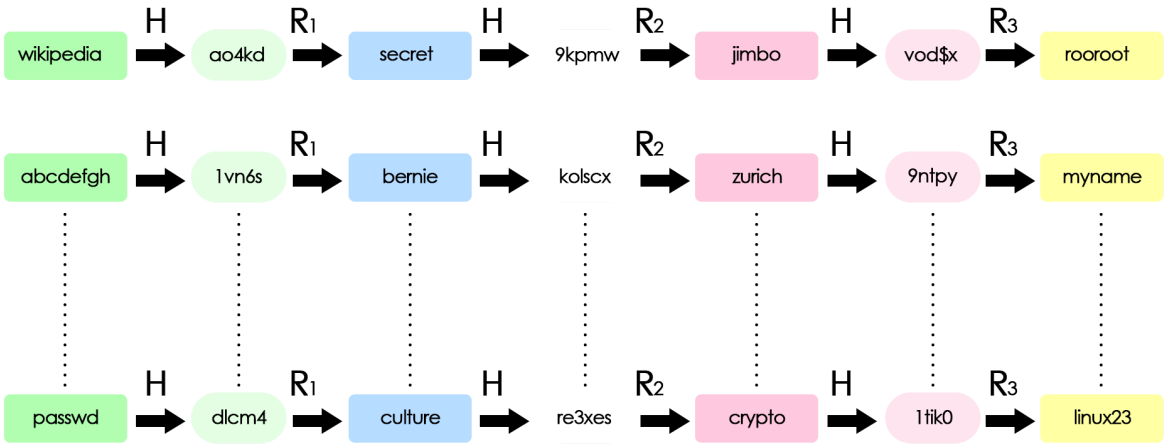
Pour notre fin de licence, nous avons réalisé un projet de cryptographie pour l'UE "Projets", sous la direction de madame BOURA, maître de conférence en Cryptographie à l'UVSQ. Ce projet est réalisé par un groupe de 4 personnes d'élèves de L3 : Théo FRATCZAK, Thomas JOLY, Thanushan PIRABAKARAN et Maya SANTINI. Le sujet choisi est celui du cassage de mot de passe basé sur des tables "Arc-en-ciel", "Rainbow table" en anglais. Nous allons détailler cette technique dans le prochain point, et expliquer tout au long de ce rapport notre approche face à ce sujet.

Les mots de passe, dans notre société, sont utilisés partout.
Autrefois, un mot de passe était un mot ou une phrase permettant de montrer à son interlocuteur qu'ils pouvaient nous transmettre un certain type de donnée généralement confidentiel.
Désormais, d'une création de compte sur un site lambda à l'identification sur un site de banque en ligne, **les mots de passe sont une nécessité pour assurer l'authentification d'une personne et l'accès à son dossier personnel et des données parfois confidentielles**. Un mot de passe doit être tenu secret pour éviter qu'un tiers, malveillant ou non, n'accède à certaines informations qui ne lui étaient pas destinées. Il doit être robuste (c'est-à-dire comporter assez de caractères, être composé de minuscules, majuscules, chiffres et caractères spéciaux) pour éviter à un potentiel hacker de le récupérer facilement : c'est ce que nous prouverons dans la suite de ce rapport.
Sur internet, des **fonctions de hachage** assurent une bonne protection de nos mots de passe. En effet, de cette manière, un attaquant n'aura pas accès aux mots de passes directement en récupérant la base de données. Elles ont pour entrée un message de taille quelconque et renvoient un message unique de taille fixe appelé empreinte, sans utiliser de clé et **permettent d'assurer l'intégrité des données**. Elles seront expliquées dans la partie sur SHA-3-256, qui est une fonction de hachage ayant des propriétés cryptographiques, comme la résistance aux pré-images qui permet de ne pas retrouver un mot de passe caché à partir d'un mot de passe donné. D'autres existent, telles que SHA-1, SHA-2, ou encore MD5. Par exemple, SHA-1 est utilisé pour la majorité des Bitcoins, et SHA-3 pour la blockchain Ethereum, ou encore Cisco, Intel et Microsoft.
Les fonctions de hachage ne sont pas utiles uniquement pour les mots de passe, mais dans bien divers protocoles comme les schémas de signature électronique tel que SSH ou comme générateurs des suites pseudo-aléatoires.

2. Présentation du projet

Le projet consiste à étudier la technique de la Rainbow Table, l'implémenter en utilisant une fonction de hachage cryptographique et de simuler une attaque dessus.

La Rainbow Table peut être schématisée ainsi :



Explicitons cette technique.

Pour commencer, un mot de passe est donné. Il va être haché une première fois, pour retourner ce qu'on appelle une empreinte ou un haché. Sur cette empreinte, nous appliquerons une fonction de réduction pour obtenir un nouveau mot de passe. Et nous répétons cette action x fois. Au final, nous retiendrons que le mot de passe original, et le mot de passe ressortant de la Rainbow Table.

La fonction de hachage (H) est identique à chaque fois, nous utiliserons SHA-3-256 pour cela, tandis que les fonctions de réduction (R) changent à chaque nœud. C'est pour cela que cette technique est appelée "Arc-en-ciel" : les mots de passes retournés après chaque fonction de réduction seront différents.
Le choix des fonctions de réduction est très important car il faut **réduire le nombre de collisions** au maximum. La collision d'une fonction de réduction est l'apparition de deux mots de passe identiques pour deux hash différents. Mais ces fonctions de réduction doivent être assez simples pour être effectuées rapidement et réduire la complexité.

L'attaque par rainbow table utilise le modèle compromis espace/temps :
l'intérêt de cette attaque comparé à l'attaque brute force est que pour l'attaque brute force, on teste chaque mot de passe un à un puis on le compare avec chaque hash possible, ce qui est très coûteux en stockage. L'attaque par rainbow table avec le paradigme espace/temps, ne reproduit qu'au maximum une seule chaîne. Plus on augmente le nombre de nœuds, plus le temps d'exécution est élevé. A l'inverse, plus on augmente le nombre de mot de passe stocké dans la table, plus la mémoire est utilisée.
Par conséquent, en gérant minutieusement la taille des nœuds et du nombre de chaînes stockées, on gagne en performance et en mémoire.

II . Déroulé du projet

1. SHA-3-256

SHA-3, *Secure Hash Algorithm*, est issu d'une compétition organisée par la NIST, National Institute of Standards and Technology, appelée *NIST hach function competition*. Elle avait pour but de trouver un nouvel algorithme qui pourrait remplacer les fonctions SHA-1 et SHA-2. L'algorithme élu par cette compétition est celui de Keccak. Actuellement, il ne remplace pas SHA-2, car aucune attaque ne l'a compromis contrairement à SHA-1, mais prévient une possible future attaque contre le standard en offrant une solution. Cependant, SHA-3 et Keccak ne sont pas identiques, car SHA-3 est la version standardisée de Keccak différant légèrement de ce dernier. Il est nécessaire de préciser que dans notre version, nous n'utilisons pas de salage, qui est une méthode de renforcement que l'on expliquera brièvement plus tard, auquel cas la Rainbow Table ne fonctionnerait pas.

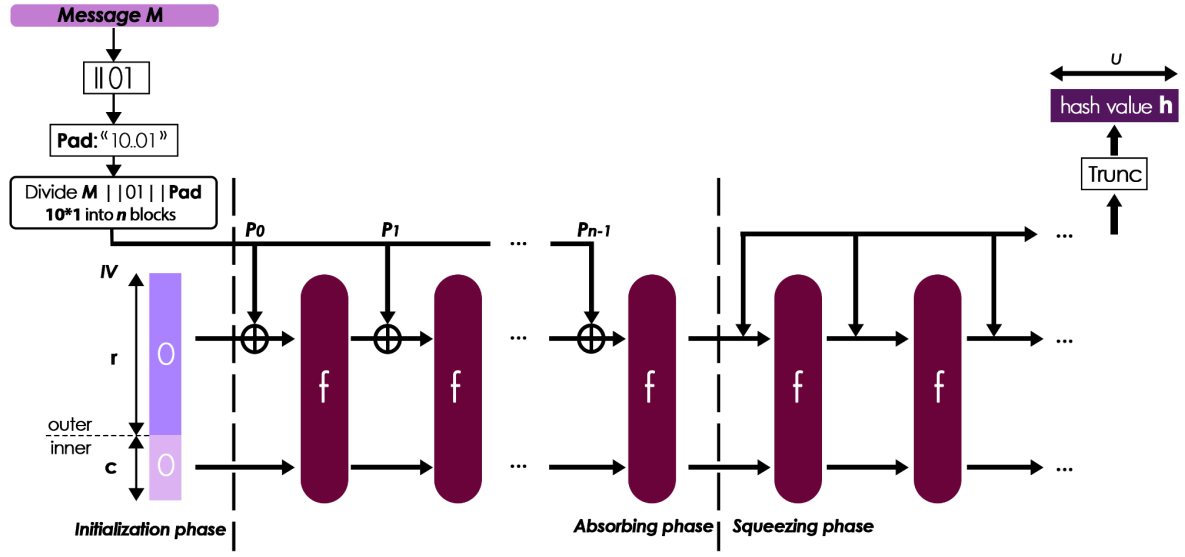
Keccak est une **fonction de hachage cryptographique** : une fonction à **sens unique**. En effet, elle est quasi impossible à inverser. Pour qu'elle soit idéale, elle devrait posséder les propriétés suivantes :

- être déterministe : le mot de passe donné aura toujours le même haché.
- le haché se calcule "facilement".
- avec un haché donné, il est impossible de retrouver son mot de passe (résistance à la pré-image).
- avec un haché donné et son mot de passe, il est impossible d'avoir un second mot de passe avec le même haché (résistance à la seconde pré-image).
- avec deux mots de passe différents, il est impossible d'avoir le même haché.
- en modifiant le mot de passe de manière minime, le haché aura un changement considérable.

Ces fonctions sont donc fréquemment utilisées pour des signatures numériques par exemple.

Keccak suit la construction dite en **éponge** : elle prend en paramètre une chaîne de n'importe quelle taille et retourne une chaîne de la longueur souhaitée. Elle opère une transformation f sur un nombre fixe b de bits. Une fonction de rembourrage p est utilisée pour préparer les blocs de message à hacher.

Voici un schéma explicitant Keccak appliqué à SHA-3 :



IV: Initial Value, **r**: rate **c**: capacity, **f**: function

Dans un premier temps, nous allons injecter notre message dans une fonction **padding**, soit bourrage en français, dans laquelle nous allons bourrer le mot :
on inverse les valeurs binaires des caractères du mot de passe en ASCII, qui est transformé en entier (int). La valeur binaire 110 est ajoutée directement à la suite, puis une suite composée de 0 uniquement, jusqu'à atteindre la longueur souhaitée et le bit de poids faible de ce nouveau binaire sera un 1.
Nous aurons un registre de taille 1600 bits sur lesquels s'opérera Keccak. Pour SHA-3-256, la longueur voulue ici est de 1088 bits. On aura donc r = P0 = 1088 bits et c = 512 composé de 0. Après le padding, nous obtiendrons donc des blocs de messages Pn. Notre mot de passe sera trop court et nous aurons donc un seul message P0.
Le résultat sera stocké sous forme de tableau, et le message, si supérieur à 64 bits, sera stocké colonne par colonne plutôt que par ligne.

Nous passons ensuite à la phase d'**absorption**.
Au départ, r sera initialisé à 1088 bits de 0. Puis, on XOR r avec P0. Le terme xoré sera donc injecté par la suite dans la fonction f de Keccak.
La fonction Keccak est définie comme une transformation sur une matrice de 5x5 entiers de 64 bits chacun, correspondant au terme xoré. Cette transformation utilise une série d'opérations non linéaires, y compris des permutations et des substitutions, qui sont appliquées à la matrice à plusieurs reprises. À chaque tour, les valeurs de la matrice sont modifiées en fonction des autres valeurs de la matrice et des valeurs du bloc de message actuel.

La fonction Keccak est conçue pour être résistante aux attaques cryptographiques en utilisant plusieurs techniques de sécurité, telles que la diffusion, la confusion, et la permutation. Ces techniques garantissent que chaque bit de la sortie dépend de chaque bit de l'entrée, rendant la fonction très robuste contre les attaques.

Celle-ci est composée de 24 tours, et chaque tour se divise en 5 phases :

- ➔ fonction theta θ
- ➔ fonction rho ρ
- ➔ fonction pi π
- ➔ fonction chi χ
- ➔ fonction iota ι

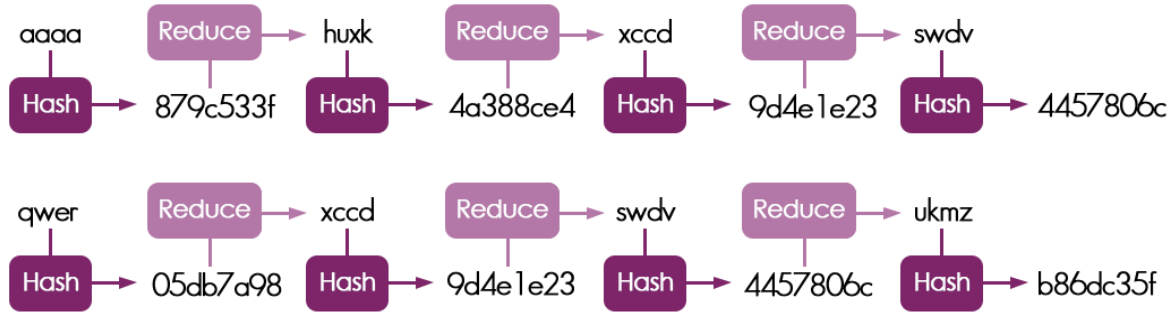
Ces fonctions seront explicitées de manière informatisée dans la partie *Implémentation*.

Après avoir passé les 24 tours, nous obtenons un haché de taille 1600 bits, duquel nous extrairons uniquement 256 bits grâce à la phase d'**essorage**. Ces derniers sont ceux qui seront donnés à la fonction de réduction qui suit.

2. Fonctions de réduction

Une fonction de réduction permet de **passer d'un haché à un nouveau mot de passe**. Dans une rainbow table, elles doivent être différentes dans une même chaîne pour éviter les collisions.

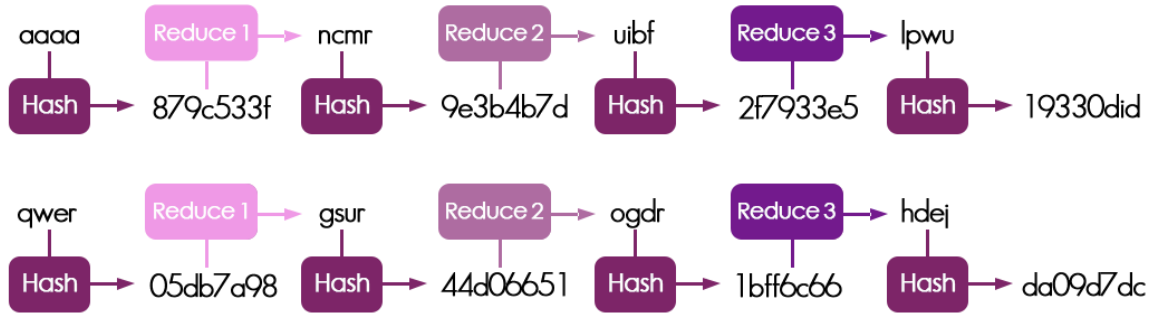
Par exemple :



Rainbow Table

aaaa	4457806c
qwer	b86dc35f

Nous pouvons observer ici la **collision** sur xccd car les fonctions de réduction sont identiques, elle a été générée par deux empreintes différentes, et on obtiendra donc une duplication de la chaîne par la suite. Pour minimiser les collisions, il faut donc avoir des fonctions de réduction différentes :



Rainbow Table

aaaa	19330did
qwer	da09d7dc

Le but est d'avoir des fonctions de réduction simples et pouvant être générées en masse. Pour cette raison, nous avons décidé de faire **une fonction XOR et d'utiliser des nonces**.

Cette fonction consiste à prendre un haché et à le XOR avec une variable nonce prédéfinie. Cette variable sera différente pour chaque fonction de réduction *(grâce à la nonce incrémentée dans le nœud qui la précède)*, bien que celles-ci consistent fondamentalement à faire la même chose, c'est à dire XOR le haché avec une nonce, octet par octet jusqu'à avoir atteint la taille du mot de passe.

Ainsi, la fonction XOR répond aux critères d'une fonction de réduction : simple et produite facilement en masse.

Nous avons effectué beaucoup de tests pour trouver la fonction de réduction la plus optimale, lesquels sont expliqués dans la partie Tests effectués.
Dans notre cas, plus la chaîne est longue, plus le risque de collision est grand dû à notre fonction de réduction qui ne prend en compte qu'une partie de la table ASCII.

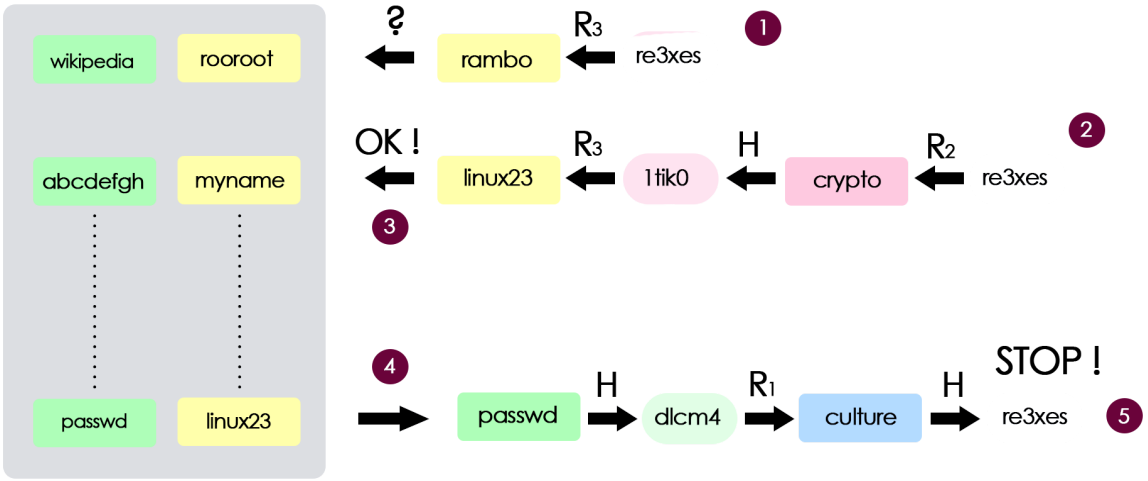
3. Attaque

Premièrement, nous allons **générer une rainbow table**, autrement dit une liste stockant des tuples contenant les premiers et les derniers éléments de chaque chaîne.
Ensuite, nous allons **créer une chaîne d'attaque** : on récupère le hash à casser, et sur lequel nous appliquons la fonction de réduction Rn-1.
Nous **parcourons les seconds éléments des tuples** dans notre rainbow table afin de trouver l'élément réduit que nous venons de générer :

- Si le résultat correspond à un mot de passe final dans notre dictionnaire, nous retournons le mot de passe originel associé.
- Si l'élément est trouvé dans la rainbow table, nous allons recréer la chaîne en partant cette fois-ci du premier élément du tuple, afin de vérifier si la chaîne d'attaque est présente dans la chaîne de la rainbow table. Autrement dit, nous réitérons l'opération avec (n-1)-1. Si c'est le cas, l'attaque est un succès.

Nous nous arrêtons lorsque le mot de passe originel est retrouvé, il est donc possible d'arriver à R0, ou ne pas retrouver le bon mot de passe si notre liste ne le contient pas.

Voici un schéma explicatif :



La rainbow table générée sera stockée sous un fichier JSON que nous réutiliserons pour chaque attaque. Nous avons réfléchi à un compromis espace/temps, et après plusieurs tests explicités dans la partie Tests effectués, nous en sommes arrivés à la conclusion qu'il faudrait **1109756 mots de passe générés et 22195 nœuds pour un mot de passe de taille 5**. Notre raisonnement est expliqué dans la section Direction prises.

Voici un tableau de comparaison pour bien comprendre le gain de mémoire avec le compromis espace/temps face à une attaque par force brute :

Comparaison de la taille du fichier avec une attaque par force brute et une attaque avec un compromis espace/temps en fonction du nombre de caractères du mot de passe

Attaque / Taille du mot de passe	Force brute	Compromis espace/temps
2 caractères	108 Ko	12 Ko
3 caractères	4,2 Mo	92 Ko
4 caractères	164 Mo	740 Ko
5 caractères	6.5 Go	5.1 Mo

4. Implémentation

L'implémentation a été faite en Rust car Python n'offre pas assez de performance et n'est donc pas adapté pour ce type d'attaque. Abordons dans le même ordre que précédemment notre code. Notons que notre implémentation ne permet pas d'effectuer les rainbow table avec un mot de passe de plus de 5 caractères avec l'alphabet suivant : les chiffres de 0 à 9, tous les caractères minuscules et les caractères spéciaux ?!#. Cela est facilement modifiable mais c'est un parti pris de notre part. Le code est entièrement commenté. Nous avons également implémenté une interface nous permettant de lancer une attaque en donnant le hash et la taille du mot de passe.

Premièrement, **le hachage avec SHA-3-256**, dans le fichier **sha3.rs** :

`fn padding(password : &str) -> u64` : Permet d'inverser les valeurs de chaque caractère, et d'appliquer le pattern voulu à la suite. Prend le mot de passe en string en argument, et renvoie un entier.

`fn bit_to_tab(m:u64) -> [[u64;5];5]` : Prend le padding (int) et le transforme en tableau de 5x5 avec pour chaque élément un u64. On obtient donc un tableau en 2 dimensions.

`fn keccak(mut a : [[u64;5];5] ) -> [[u64;5];5]` : Lance l'algorithme de Keccak à 24 tours avec un appel à la fonction `round`. Prends le tableau de dimension 25 et retourne un tableau de même dimension.

`fn round(mut a : [[u64;5];5], v_rc : u64) -> [[u64;5];5]` : Appelle les fonctions `theta`, `rho`, `pi`, `chi` et `iota`. Prends le tableau de dimension 25 et retourne un tableau de même dimension.

`fn theta(mut a : [[u64;5];5]) -> [[u64;5];5]` : A est la matrice (tableau) d'entrée. On xor toutes les colonnes de A entre elles pour ne retourner qu'une colonne stockée dans C. On xor ensuite chaque ligne de C avec une rotation de C, et stockons le résultat dans D. Enfin, on xor chaque élément de A avec D. On retourne A modifié.

`fn rho_pi(a : [[u64;5];5]) -> [[u64;5];5]` : On associe à une nouvelle matrice B les valeurs de A, matrice d'entrée, décalées. On retourne B.

`fn chi(mut a : [[u64;5];5], b : [[u64;5];5],) -> [[u64;5];5]` : On prend les éléments communs de l'inverse du B de la ligne +1 et colonne actuelle et du B de la ligne +2 et colonne actuelle. Puis on xor le tout avec B de la ligne et colonne actuelle. On fait cela pour toutes les colonnes, stocke le résultat dans A et on retourne A.

`fn iota(mut a : [[u64;5];5], v_rc :u64) -> [[u64;5];5]` : On xor la première valeur de A avec V_RC, qui change selon le tour auquel on est. On retourne A.

`fn extraction(big_tab:[[u64;5];5]) -> [u8;32]` : Prend le tableau renvoyé par keccak, tronque les 256 premiers bits : chaque octet est inversé, puis on en fait un tableau et on le renvoie.

`pub fn sha3(password : &str) -> [u8;32]` : Prend un mot de passe et renvoie le haché. Applique l'algorithme de SHA-3-256 dans le bon ordre. La fonction est en publique pour l'appeler dans d'autres fichiers.

Les fonctions de réduction, dans le fichier **reduction.rs** :

`pub fn reduce_xor(hash: [u8; 32], nonce: u32) -> String` : Prend en argument le haché et la nonce, et renvoie le nouveau mot de passe. Cette fonction va, grâce à la nonce, changer à chaque fois. On s'arrête de xor lorsqu'on a xor autant d'octets qu'il y a dans le mot de passe.

`fn to_password(bytes: &[u8; 32]) -> String` : Prend en argument le haché qui a subi la fonction de réduction, puis tronque en fonction de la taille du mot de passe et le renvoie en chaîne de caractères (string).

La rainbow table, dans le fichier **rainbowtable.rs** :

`fn generate_table(startpassword: u32, endpassword: u32,starting_items_shared: Arc<Mutex<Vec<String>>>, bar:Arc<Mutex<ProgressBar>>) -> Vec<Node>` : Génère une rainbow table et renvoie un tableau, composé d'une structure "nœud" dont la première valeur sera le début de la liste chaînée, et la deuxième la dernière. Appelle `contains` pour générer un start différent pour chaque nœud.

`fn contains(elt:String, tab: &mut Vec<String>) -> bool` : Vérifie si un élément est dans un vecteur. Ici, utilisé pour vérifier que chaque start n'a pas déjà été utilisé en start d'une autre ligne de la table.

`fn pool() -> Vec<Node>` : Génère des threads pour permettre la parallélisation de la fonction `compare_end`, puis fusionne les tableaux créés par les threads en une seule et renvoie cette nouvelle table.

L'attaque, dans le fichier **attack.rs** :

`pub fn execution(rainbow_table: &mut Vec<Node>, hash_flag: [u8; 32]) -> bool` :

Récupère la table rainbow table sur laquelle exécuter l'attaque et le hashé donné par le mot de passe que l'on veut retrouver, puis retourne un booléen en fonction du succès de l'attaque.

L'attaque fonctionne de la manière suivant :

- Étape 1 : A partir du hash du flag, on applique la dernière fonction de réduction utilisée par la rainbow table (R_n), puis on compare cette valeur réduite aux mots de passe de fin de chaîne stockés dans la rainbow table.

- Étape 2 : Il y a deux cas possibles: si le mot de passe n'est pas présent dans la rainbow table, on applique n fois l'étape 1 en utilisant les fonctions de réduction n-1. Dans le cas contraire, si le mot de passe est présent dans la table, on va appliquer l'étape 3.
- Étape 3 : On arrive à l'étape 3 lorsqu'on a une chaîne potentielle à vérifier. En effet il y a des collisions dans les rainbow table donc il faut s'assurer que la chaîne générée est la bonne. Pour cela, on va simplement recréer la chaîne depuis le mot de passe start, et vérifier si la chaîne générée dans l'étape 2 se trouve dans cette nouvelle chaîne. Si elle est présente, l'attaque est réussie et le mot de passe est la dernière valeur réduite de la chaîne de l'étape 2.

`fn compare_end(rainbow_table: &mut Vec<Node>, value: String, start: u32, end: u32) -> Vec<u32>` : Cherche si une valeur passée en paramètre est présente dans la partie "end" de la rainbow table. Stocke toutes les positions possibles de la valeur dans un vecteur.

`fn reverse(rainbow_table: &mut Vec<Node>, hash_flag: [u8; 32]), position_flag: u32) -> bool` : Correspond à l'étape 3 de l'attaque, consistant à recréer la chaîne, pour vérifier si l'empreinte de la chaîne d'attaque est bonne.

`fn pool_search(rainbow_table: &mut Vec<Node>, value: String) -> Vec<u32>` : Génère des threads pour permettre la parallélisation de la fonction `compare_end`, puis fusionne les vecteurs créés par les threads en une seule et renvoie ce nouveau vecteur.

Enfin, le fichier **main.rs** qui permet de lancer le code correctement en appelant les fichiers nécessaires.

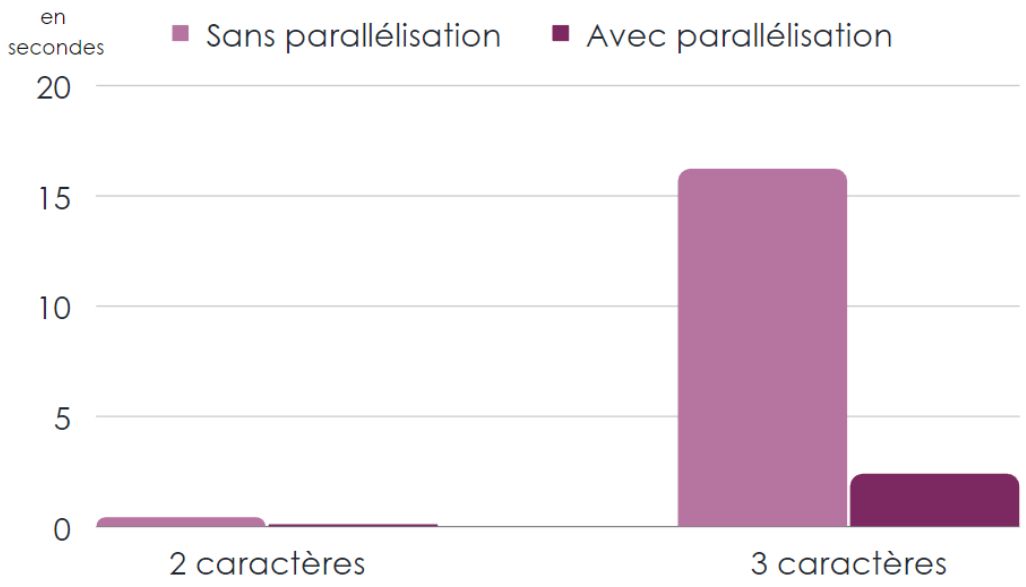
Nous avons également les fichiers **test.rs** comprenant les tests unitaires, **constants.rs** regroupant toutes les constantes, **performance.rs** pour calculer la performance, **file.rs** pour conserver la table crée dans un fichier JSON et pouvoir la réutiliser par la suite, **lib.rs** pour regrouper les fichiers et **interface.rs** permettant d'avoir une interface. Un fichier **readme.md** sera disponible pour expliquer le fonctionnement du lancement des commandes.

III . Difficultés rencontrées

1. Performance

La performance de notre code a été améliorée grâce à la parallélisation. En effet, nous avons utilisé une librairie nommée rayon qui nous a permis d'utiliser des threads et de **paralléliser la création de table et l'attaque**. Par exemple, pour 3 caractères, 480 mots de passe et 250 nœuds, la création de table s'effectue en 16 secondes sans la parallélisation, et en 2 secondes avec la parallélisation *(avec 12 threads)*. Le code a été optimisé de façon à s'adapter au nombre de cœurs que l'ordinateur qui l'exécute possède. Voici un graphique illustrant le gain de temps avec la parallélisation :

Comparaison du temps d'exécution avec et sans parallélisation en fonction du nombre de caractères du mot de passe avec 12 threads

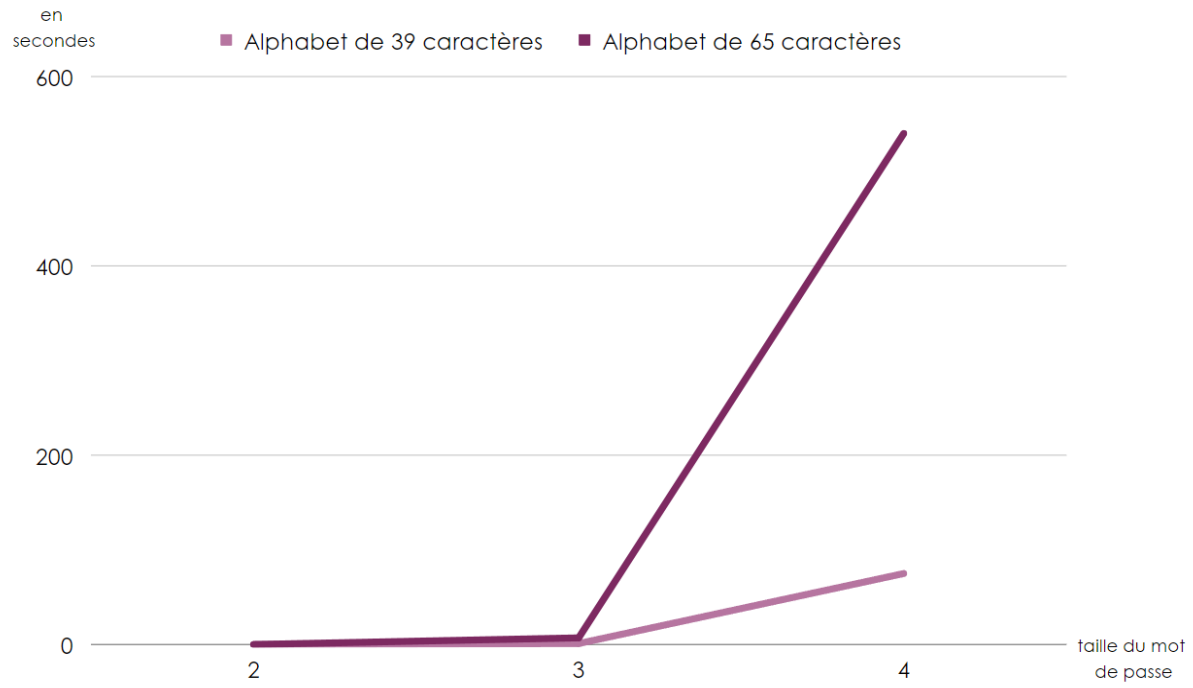


De plus, **nous stockons la génération de la rainbow table** dans un fichier JSON pour pouvoir la réutiliser pour les attaques sans devoir la régénérer, ce qui nous fait gagner beaucoup de temps étant donné la grande quantité de mots de passes qu'elle recouvre.
A noter également que pour éviter de devoir faire des décalages et des permutations sur un tableau de 64 éléments (dans SHA-3), ce qui aurait été moins performant, nous avons implémenté de simples opérations bits à bits tels que des masks, shift ou xor.

2. Directions prises

La fonction de réduction choisie découle du rendez-vous avec madame BOURA. Nous avions plusieurs idées erronées et elle nous a permis de nous recentrer et de faire le bon choix. Nous voulions faire dans un premier temps seulement 4 fonctions de réduction différentes assez simples, puis nous avons compris assez rapidement l'importance de la nonce pour générer beaucoup de fonctions de réduction simplement et rapidement. La performance de cette fonction est correcte et les collisions ne sont pas nombreuses. Nous avons effectué beaucoup de tests pour trouver la fonction de réduction la plus efficace.
Concernant le nombre de caractères choisis, nous avons opté pour 5 caractères. En effet, 6-7 caractères prendraient trop de temps à être exécutés, sachant que dans le mot de passe, nous pouvons utiliser des minuscules, chiffres et caractères spéciaux (?!#). En dessous de 5 caractères, c'est un peu court. Nous avons donc choisi ce compromis ; pour 5 caractères, nous aurons **39^5 possibilités de mots de passe**. Les majuscules ont été omises car nous passerions de 39^5 possibilités à 65^5 possibilités, qui est un nombre trop grand pour nos ressources. Voici un graphe illustrant le temps d'exécution pour les différentes possibilités pour 3 tailles de mot de passe:

Comparaison du temps d'exécution en secondes pour la création d'une rainbow table entre un alphabet de 39 et de 65 caractères selon le nombre de caractères du mot de passe



Nous avons par ailleurs décidé pour SHA-3 d'utiliser des u64 dans la matrice de 5x5 contrairement à ce que le site officiel de keccak conseillait de faire.
Quant au compromis espace/temps, nous nous sommes aidés du test de pourcentage que nous avons créé et expliqué dans la section Tests effectués. Il fallait trouver une solution pour trouver le juste milieu entre l'utilisation de la mémoire et le temps d'exécution. Nous avons donc décidé de chercher le bon ratio qui nous permet de générer une rainbow table

contenant entre 85% et 90% des mots de passe possibles. Au-dessus, l'exécution devenait beaucoup trop longue, et en dessous, les chances de ne pas retrouver les mots de passe augmentaient.
Cependant il est quand même possible de ne pas retrouver le mot de passe que l'on cherche dans nos rainbow table. Pour pallier à ce problème, nous avons choisi de laisser la possibilité à l'utilisateur de créer jusqu'à 3 versions différentes d'une rainbow table et d'effectuer des attaques sur chacune d'entre elles. De ce fait, la probabilité de ne pas retrouver un mot de passe diminue drastiquement.

3. Difficultés techniques

Durant ce projet, nous avons dû faire face à plusieurs obstacles.
Tout d'abord, l'implémentation de SHA-3-256. La documentation sur internet n'étant pas abondante, nous avons dû nous débrouiller avec ce que nous avions à disposition. Par exemple, sur la page Wikipédia de Sha-3-256, certaines informations sont omises. De ce fait, nous avons eu des **problèmes d'inversion** qui n'étaient pas précisés, ce qui nous a retardé jusqu'à trouver le souci nous-même en testant.
De plus, en Rust, le message était trop long pour être inscrit dans un u128 (entier non signé codé sur 128 bits), nous avons donc dû faire un tableau d'octets de 32 éléments.
Les tests sur des mots de passe pour trouver le meilleur compromis espace/temps de plus de 4 caractères étaient longs, c'est pourquoi nous avons pu les tester en grande quantité jusqu'à 4 caractères, composés de minuscules et chiffres uniquement.
En outre, nous voulions savoir quel pourcentage de mot de passe nous testions avec notre rainbow table afin de trouver des valeurs pertinentes pour la taille de cette dernière. La fonction que nous avons créé nous permettant de connaître le pourcentage de mot de passe testé est très longue à l'exécution, bien que nous l'ayions parallélisé, ce qui a rendu les tests sur des rainbow tables dont la taille des mots de passe était supérieure à 4 caractères plus compliqués à faire. Nous préciserons ce test dans la partie Tests effectués. Nous avons trouvé une formule mathématiques nous donnant une approximation des meilleurs paramètres à avoir pour la rainbow table.

IV . Conclusion

1. Tests effectués

Nous avons dans un premier temps testé nos codes avec des vecteurs tests trouvés sur internet, et notre attaque sur un mot de passe de 2 caractères composé de caractères minuscules. Puis, nous avons ajouté la possibilité des chiffres de 0-9, et des caractères spéciaux ?!#. Puis nous avons augmenté la taille du mot de passe jusqu'à atteindre 5 caractères.

Nous avons créé des **tests unitaires** pour SHA-3-256, l'attaque et la fonction de réduction qui fonctionnent pour une valeur donnée. Puis, de notre côté, nous avons créé plusieurs autres tests.

Par exemple, un **test permettant de savoir le pourcentage de mots de passes qui ont été générés par rapport au nombre de mots de passes total possibles**. Par exemple, pour 2 caractères minuscules/chiffres, on peut facilement atteindre les 100%.

Pour un mot de passe de 3 caractères minuscules/chiffres, 500 lignes et 200 noeuds, et en changeant quelle fonction de réduction on utilise, on obtient les pourcentages suivants :

cf Annexe pour les valeurs des tests trouvées

C'est pourquoi, la fonction xor est la meilleure.

Pour 3 caractères, 480 lignes et 250 nœuds semble être le meilleur ratio possible (81.9%).

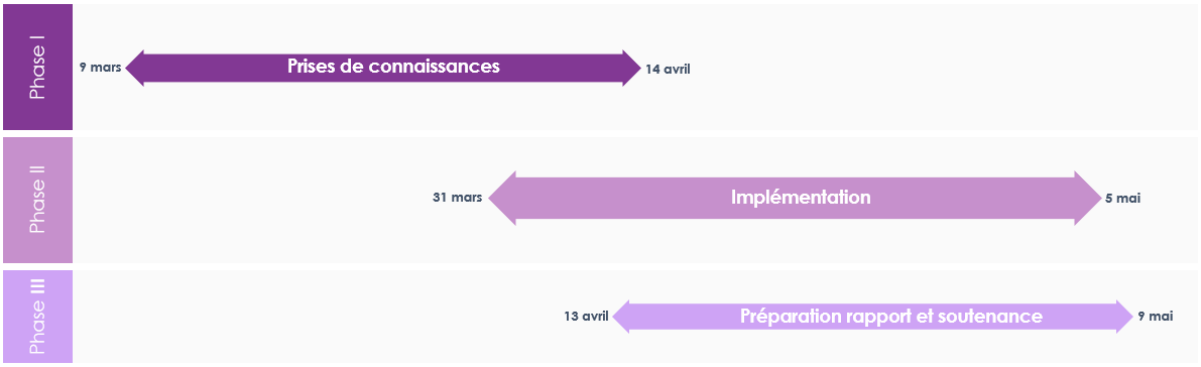
Pour 4 caractères, 20990 lignes et 400 nœuds nous donnent un pourcentage élevé (90%). Le test a duré 1h15. Le fichier JSON stocké faisait 615 Ko.

Suite à cela, nous avons réfléchi au meilleur compromis espace/temps puisque nous ne pouvions pas nous permettre de lancer des tests de plus d'une heure sans logique. Nous avons alors trouvé une relation semblant nous donner des résultats oscillant entre 85% et 90%, avec nombre de lignes m, nombre de noeuds n, nombres de caractères dans le mot de passe c et taille de l'alphabet a :

$$m = \sqrt{\left(\left((c + 1) \times a^c\right) \times 50\right)} \quad \text{et } n = \frac{m}{50}$$

Nous aurons donc pour 5 caractères 164521 lignes et 3290 nœuds.

2. Evolution, limites et conclusion du projet



Au cours de ce projet, nous avons dû faire des choix et décider de laisser de côté certaines options.

Par exemple, nous n'avons pas inclus les majuscules dans l'alphabet des mots de passe, ce qu'on aurait pu faire. Cela multiplie de manière trop significative le nombre de possibilités de mots de passe possibles, mais le fait de ne pas l'avoir inclus nous écarte du réalisme des mots de passe utilisés sur les sites et nous fait comprendre que la complexité de la sécurité des mots de passe est bien plus haute. De plus, nous aurions pu augmenter le nombre de caractères de notre mot de passe. En effet, 5 caractères reste un chiffre très faible, la sécurité du mot de passe le sera aussi en conséquence. Aujourd'hui, tous les sites vont demander minimum 8 caractères avec un alphabet plus étendu.

En outre, nous aurions pu donner en option à l'utilisateur de lancer l'attaque sur tous les mots de passes entre 2 et x caractères, plutôt que de devoir préciser le nombre de caractères voulus.

Sans oublier le salage. C'est une méthode de renforcement qui n'est pas à notre portée et qui permet de complexifier les fonctions de hachages. Avec le salage, il aurait été plus compliqué pour nous d'attaquer avec une rainbow table. Mais, encore une fois, cela nous écarte de la réalité où le salt/pepper est utilisé partout. Le salt est une valeur connue ajoutée au mot de passe avant hachage, tandis que le pepper est secret, il peut être conservé à part du haché, ou pas conservé du tout.

Quant à la fonction de réduction, nous aurions pu procéder autrement et faire en sorte d'avoir moins de collisions en prenant en compte la table ASCII complète par exemple.

En conclusion, nous avons fait notre possible à notre échelle mais nous aurions pu changer des aspects tels que le nombre de caractères, l'alphabet du mot de passe et la fonction de réduction utilisée. Nous sommes encore loin des systèmes actuels de protection des mots de passe, mais nous avons pu en connaître une partie ce qui nous a beaucoup plu.

IV . Annexes

[p15]

Fonction xor hash^nonce : 79.71751% ~ 81.97445% *(480 mots de passe et 250 noeuds)*
Fonction mix xor : 76.26457% ~ 77.95139%
Fonction nonce - un octet de haché : 77.194786%
Fonction xor et index-nonce^hash : 77.1262%
Fonction xor index*nonce^hash : 77.09619%
Fonction double xor et nonce/index+1 : 76.896866%
Fonction fusion hash et nonce mélangés : 76.77683%
Fonction xor où res = nonce >> i; res^hash : 76.56679%
Fonction xor nonce+index^hash : 76.55821%
Fonction xor nonce/index^hash : 76.53464%
Fonction tronque + xor : 76.228134%
Fonction not xor et addition du nonce : 76.20885 %
Fonction double xor et nonce*index^hash pour le 2eme : 76.1124%
Fonction double xor et !nonce*index^hash pour le 2eme : 75.99237%
Fonction xor index+nonce^hash : 75.84448%
Fonction and hash + nonce + index : 75.36437%
Fonction and hash + nonce : 75.252914%
Fonction modulaire : 75.22291%
Fonction double xor et nonce-index : 61.031807%
Fonction double xor et nonce+index : 59.531464%
Fonction double xor : 9.034207%
Fonction tronque : 8.800583%
Fonction double xor et not nonce : 4.4731655%
Fonction double xor puis ou : 2.2955246%

Sources :

https://infosecwriteups.com/breaking-down-sha-3-algorithm-70fe25e125b6
https://keccak.team/keccak_specs_summary.html
https://fr.wikipedia.org/wiki/SHA-3
https://fr.wikipedia.org/wiki/Fonction_de_hachage_cryptographique
https://csrc.nist.gov/projects/hash-functions/sha-3-project
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
https://assiste.com/Securite_des_mots_de_passe/Attaque_par_tables_arc-en-ciel.html
https://emn178.github.io/online-tools/sha3_256.html
https://kerkour.com/multithreading-in-rust
https://www.di-mgt.com.au/sha_testvectors.html
https://eprint.iacr.org/2017/190
https://mieuxcoder.com/2008/01/02/rainbow-tables/
https://boowiki.info/art/methodes-de-cryptanalyse/tableau-arc.html
https://simplicable.com/IT/salt-vs-pepper