Quentin Gavilan

MS-SIS

Projet programmation C

Rain-C

2025-2026

I. À quoi sert le programme ?

Rain-C est un utilitaire écrit en C dont la fonction est de générer et interroger des tables de correspondance entre condensats (hash) et mots de passe, appelées T3C (Table de Correspondance).

Il offre un outil pédagogique et open-source permettant d'expérimenter les mécanismes de **Rainbow tables**, afin d'illustrer de manière simple et reproductible les bases du hachage et de la récupération de mots de passe.

En pratique, Rain-C permet de :

- **Créer** un fichier .t3c à partir d'un dictionnaire avec un mot de passe par ligne en calculant pour chaque mot son condensat à l'aide de l'un des algorithmes *SHA-256, SHA-512, BLAKE2b-512 ou SHA3-256* (via la bibliothèque OpenSSL), puis en stockant les paires *condensat<TAB>motdepasse*.
- **Charger** une table T3C existante en mémoire, l'indexer à l'aide d'un arbre binaire de recherche et retrouver rapidement le mot de passe clair correspondant à un condensat fourni (option -s ou flux stdin).
- **Suivre** l'avancement des opérations grâce à des barres de progression, particulièrement utiles lors du traitement de dictionnaires volumineux.

II. Objectifs du projet

Rain-C répond à plusieurs objectifs pédagogiques et techniques :

- **Implémenter en C** une logique complète de génération et de recherche de tables de correspondance d'un hash et d'une chaine de caractère.
- **Explorer les structures de données dynamiques**, notamment le BST pour un indexage rapide en mémoire.
- **Utiliser OpenSSL** pour le calcul des condensats via les algorithmes *SHA-256, SHA-512, BLAKE2b-512 et SHA3-256*.
- **Permettre le traitement en flux (stdin/stdout)** pour une utilisation flexible en ligne de commande et en pipeline.
- **Assurer la reproductibilité** grâce à un environnement **Docker** portable et facile à exécuter.

III. Gestion de projet

Pour assurer une bonne progression du développement et éviter de me disperser, j'ai mis en place une méthode de gestion de projet structurée, adaptée à ma manière de travailler et à mes contraintes de temps.

J'ai choisi d'utiliser des cartes Kanban, un outil visuel simple mais efficace pour organiser les différentes étapes du projet. J'ai structuré mon tableau autour de plusieurs colonnes clés :

- À faire : regroupe les fonctionnalités prévues, les correctifs à apporter et les tests à réaliser
- En cours : permet de visualiser les tâches sur lesquelles je travaillais
- Terminé : afin de mesurer l'avancement global et de garder une trace du travail accompli.

Cette méthode m'a permis d'avoir une vue d'ensemble claire et motivante sur le projet. À chaque session de développement, je pouvais rapidement identifier mes priorités, estimer la charge de travail et planifier la suite.

En parallèle, j'ai tenu un agenda hebdomadaire détaillant les objectifs de chaque journée : portions de code à écrire, tests à réaliser, corrections à valider. Cet agenda m'a aidé à structurer mon temps et à éviter de repousser certaines tâches complexes. Malgré quelques écarts entre la planification et la réalité, cette organisation m'a apporté une discipline de travail et une meilleure visibilité sur mes progrès.

Enfin, cette approche m'a permis de comprendre que la gestion de projet ne se résume pas à un simple planning, mais qu'elle repose sur la capacité à ajuster ses priorités, à évaluer ses limites, et à maintenir une constance dans l'effort.

IV. Structure du code

Fichier	Rôle principal
main.c	Point d'entrée du programme.
menu.c	Interface utilisateur CLI, affichage du menu, gestion des prompts interactifs. Parse les arguments, gère les modes (-G / -L).
control_T3C.c	Gestion des tables T3C : création, insertion, recherche, lecture et parsing d'un fichier t3c, gestion de l'arbre binaire.
control_dict.c	Lecture du dictionnaire, gestion du flux et génération des paires hash/mdp.
hash.c	Calcul cryptographique des condensats via OpenSSL.
include/*.h	Déclarations et prototypes des fonctions utilisé par d'autre fichier .c

Si nous rentrons dans les détails des fichiers du programme :

A. menu.c

Rôle : Ce fichier gère l'interface en ligne de commande de Rain-C. Il analyse les arguments transmis au programme et orchestre les différents modules (control_dict.c et control_t3c.c) pour exécuter les actions demandées.

Fonctions principales :

- **`help(char *prog)`**
  Affiche un guide d'utilisation détaillé avec les options disponibles, les modes de génération (-G) et de recherche (-L), les algorithmes autorisés, et des exemples.

- **`algo_exist(char *algo_name)`**
  Vérifie que l'algorithme de hachage choisi (*sha256, sha512, blake2b512, sha3-256*) est autorisé. Retourne 0 si l'algorithme existe, -1 sinon.

- **`exec_mode(void)`**
  Exécute l'action correspondant au mode choisi :

  1. **Mode génération (-G)** :
     Lit un dictionnaire et calcule les hash via *dict_to_Table()*de *control_dict.c*.
     Génère le fichier T3C via *create_t3c()* de *control_T3C.c*.

Affiche un résumé et libère la mémoire.

2. **Mode recherche (-L)** :
   Si l'utilisateur fournit un hash avec -s, recherche directe dans le T3C via *t3c_mode_lookup()*.
   Sinon, demande le hash à rechercher via stdin.

- **`start(int argc, char **argv)`**
  Analyse les arguments de la ligne de commande et initialise les variables globales (*mode, dict_path, t3c_path, algo_choice, hash_search*).
  Vérifie l'existence et la lisibilité des fichiers fournis (*dict.txt ou table.t3c*).
  Valide les options spécifiques à chaque mode.
  Appelle *exec_mode()* pour lancer l'action.

  Variables globales :
  - *mode :* -1 non défini, 0 = génération, 1 = recherche
  - *dict_path :* chemin vers le dictionnaire
  - *hash_search :* hash à rechercher si fourni
  - *t3c_path :* chemin par défaut ou fourni du fichier T3C
  - *algo_choice :* algorithme de hachage choisi (sha256 par défaut)

B. control_T3C.c

Rôle : Ce module implémente toutes les opérations liées à la table T3C : lecture, écriture, chargement en mémoire, indexation en arbre de recherche et lookup d'un condensat. C'est le cœur du moteur de recherche du programme (mode -L).

Fonctions principales :

- **`keepStr(char *str)`**
  Duplique une chaîne pour en conserver la valeur indépendamment des tampons temporaires.

- **`parse_t3c(char *ligne)`**
  Découpe une ligne hash\tmdp, nettoie les caractères parasites et renvoie un pointeur vers le mot de passe.

- **`progress_bar(size_t done, size_t total, char *text)`**
  Affiche une barre de progression dynamique pendant la lecture ou l'écriture.

- **`count_lignes(FILE *f)`**
  Compte le nombre de lignes valides dans un fichier T3C pour estimer la progression et allouer la mémoire.

- **`t3c_init()`** / **`t3c_free()`**
  Initialise ou libère complètement une table T3C en mémoire.

- **`t3c_add()`**
  Ajoute une paire (hash, mot_de_passe) à la table.

- **`create_t3c()`**
  Écrit la table complète dans un fichier *.t3c* avec un en-tête et suivi de progression.

- **`t3c_load()`**
  Charge un fichier *.t3c* en mémoire, nettoie et insère chaque entrée.

- **`t3c_index_build()`** / **`t3c_index_insert()`** / **`t3c_lookup()`** / **`t3c_index_free()`**
  Implémente la gestion complète de l'arbre binaire (construction, insertion, recherche et libération).

- **`t3c_mode_lookup()`**
  Fonction principale du mode -L :
  - Charge la table T3C
  - Construit l'index BST
  - Recherche un hash fourni ou lit des hash depuis stdin
  - Affiche le mot de passe trouvé

C. control_dict.c

Rôle : Ce module est responsable de la génération des tables T3C à partir d'un dictionnaire de mots de passe. Il lit le fichier ligne par ligne, calcule le condensat de chaque mot à l'aide des fonctions du module *hash.c*, et enregistre les couples *hash mot_de_passe* dans une structure *t3c_table*. Il constitue le cœur du mode -G du programme.

Fonctions principales :

- **`progress_bar_dict(size_t done, size_t total, char *text)`**
  Affiche une barre de progression dynamique pendant le hachage du dictionnaire.

- **`count_lignes_dict(FILE *fichier)`**
  Compte les lignes non vides d'un dictionnaire pour dimensionner la table et calibrer la progression.

- **`dict_to_Table(char *path, char *algo_name, t3c_table *table)`**
  Fonction principale de conversion :
  - Lit chaque mot du dictionnaire (ligne par ligne).
  - Supprime les caractères parasites (\r, \n).
  - Calcule le condensat via *string_to_hash()* du module *hash.c*.
  - Convertit le résultat binaire en hexadécimal avec *bin_to_hex()*.
  - Ajoute la paire (hash, mot_de_passe) à la table via *t3c_add()*.
  - Met à jour la barre de progression.

D. hash.c

Rôle : Ce module encapsule toutes les opérations de hachage cryptographique utilisées par Rain-C. Il s'appuie sur la bibliothèque OpenSSL pour garantir la sécurité, la portabilité et la compatibilité avec les algorithmes (*SHA-256, SHA-512, BLAKE2b-512, SHA3-256*). Il fournit deux fonctions essentielles : le calcul du condensat et sa conversion en format hexadécimal lisible.

Fonctions principales :

- **`int string_to_hash(char *input, char *algo_name, unsigned char digest[EVP_MAX_MD_SIZE], unsigned int *digest_taille)`**
  Calcule le condensat d'une chaîne input avec l'algorithme spécifié (algo_name).
  Étapes :
  1. Récupère le descripteur de l'algorithme (*EVP_get_digestbyname*).
  2. Initialise un contexte de calcul (*EVP_MD_CTX_new, EVP_DigestInit_ex*).
  3. Alimente le calcul avec la chaîne d'entrée (*EVP_DigestUpdate*).
  4. Termine le hachage et récupère le digest binaire + sa taille (*EVP_DigestFinal_ex*).
  5. Nettoie ensuite la mémoire allouée au contexte (*EVP_MD_CTX_free*).
  6. Retourne 0 en cas de succès, -1 en cas d'erreur.

- **`void bin_to_hex(unsigned char *digest_bin, unsigned int digest_taille, char *digest_hex)`**
  Convertit le digest binaire obtenu précédemment en une représentation hexadécimale.
  Chaque octet est transformé en deux caractères hexadécimaux.
  Le résultat est une chaîne terminée par \0, directement exploitable dans les fichiers T3C.

V. Utilisation du programme

Les instructions détaillées d'exécution sont disponibles dans le README du dépôt GitHub, qui décrit pas à pas la compilation, l'utilisation et l'exécution via Docker. Voici le lien vers le gitHub : https://github.com/gaviepro/Rain-C.git

En résumé, Rain-C s'utilise en ligne de commande et propose deux modes :

- **Génération (-G)** : création d'une table T3C (hash ↔ mot de passe) à partir d'un dictionnaire texte.
- **Recherche (-L)** : interrogation d'une table existante pour retrouver le mot de passe correspondant à un condensat.

Exemple :

```
./lab/rainc -G lab/rockyou_1000.txt -o lab/rainbowTAB.t3c -a sha256

./lab/rainc -L lab/rainbowTAB.t3c -s
496b7706d1f04a4d767c41175895960261100 67cf81fda050056a0460b03e109
```

`-o` : spécifie le fichier de sortie de la table (par défaut *lab/rainbowTAB.t3c*)

`-a` : définit l'algorithme de hachage parmi *sha256, sha512, blake2b512, sha3-256* (par défaut sha256)

`-s` : recherche un condensat spécifique ; sinon, l'entrée peut être lue depuis stdin.

Le README fournit également :

- Les commandes Docker pour exécuter Rain-C dans un environnement isolé et reproductible (Linux & Windows).
- Les étapes de compilation locale via *make* sur Debian/Ubuntu.

Ainsi, l'utilisateur peut choisir entre une exécution native ou un déploiement sur Docker, selon ses besoins et sa plateforme.

VI. Difficultés rencontrées

Au cours du développement de ce projet, j'ai été confronté à plusieurs difficultés qui m'ont amené à faire des choix stratégiques afin d'assurer la réussite du programme.

A. Mise en place de l'environnement de développement

La première difficulté concernait le choix et la mise en place de l'environnement de développement. Ne disposant que d'un ordinateur sous Windows, il m'a fallu trouver une solution adaptée pour programmer efficacement en C, un langage historiquement mieux pris en charge sous Linux.

Souhaitant conserver Visual Studio Code, mon IDE habituel, j'ai cherché un moyen d'allier son confort d'utilisation à la compilation GNU (gcc). Après avoir analysé mes besoins :

- Accès à un compilateur fiable,
- Possibilité d'utiliser des bibliothèques externes
- Gestion propre des répertoires sources

J'ai décidé de mettre en place un **environnement hybride**. Concrètement, j'ai déployé une machine virtuelle Linux sur mon homelab, hébergeant les fichiers sources du projet. J'ai ensuite monté ce répertoire comme lecteur réseau sur Windows, me permettant de coder directement depuis VS Code tout en compilant et exécutant le programme à distance via SSH.

Ce choix m'a permis de travailler dans un environnement stable, reproductible et conforme aux besoins du projet, tout en optimisant mon temps et mes outils.

B. Organisation et planification du projet

Par la suite, comme expliqué précédemment, j'ai mis en place une gestion de projet rigoureuse en m'appuyant sur des cartes Kanban, ce qui m'a permis d'avoir une vision claire de l'avancement global et de suivre précisément les tâches restantes.

Cependant, l'une des difficultés majeures a été l'organisation. Développer un projet de cette envergure peut facilement pousser à se disperser, à s'éparpiller dans plusieurs directions à la fois, et parfois à ressentir une certaine démotivation face à l'ampleur du travail à accomplir. Les listes de tâches m'ont aidé à garder un cap clair et à maintenir un suivi cohérent du projet, mais la véritable difficulté résidait dans la rigueur quotidienne.

L'agenda que je m'étais fixé en début de semaine n'a pas toujours pu être respecté jusqu'au bout, ce qui m'a fait perdre un temps précieux que j'ai dû rattraper sur mes heures de repos ou de loisirs.

Cette expérience m'a appris à mieux équilibrer planification et flexibilité, tout en restant persévérant face aux imprévus et à la charge de travail.

C. Difficultés techniques et compréhension des structures de données

Une fois l'environnement stabilisé, les principales difficultés se sont déplacées vers la partie technique du projet.

Certaines fonctions, notamment celles du fichier *control_t3c.c*, ont nécessité un travail approfondi de compréhension des structures de données et des mécanismes de contrôle entre les condensats et les mots de passe. La mise en œuvre de l'arbre binaire de recherche a également été un point délicat, demandant de nombreux tests et recherches complémentaires pour garantir un fonctionnement correct et une gestion mémoire propre.

Les fonctions *t3c_free* et *t3c_index_free* ont particulièrement posé un problème. Lors des séances de cours, je n'avais pas réussi à obtenir un passage complet sous Valgrind, l'outil de détection de fuites mémoire. J'ai d'abord tenté plusieurs versions itératives, mais leur complexité dépassait mon niveau initial en C me créant plus de problème que de solution.

C’est finalement après avoir échangé avec un camarade que j’ai trouvé la solution : transformer ces fonctions en implémentations récursives, beaucoup plus simples et adaptées à la structure de données utilisée. Cette approche a non seulement résolu mes erreurs mémoire, mais m’a aussi permis de mieux comprendre la puissance de la récursion en C.

D. Difficultés techniques et compréhension des allocations mémoire

Le segmentation fault cette erreur légendaire, à la fois redoutée et frustrante, qui n’apporte aucune explication claire sinon celle-ci : “ton code n’est pas correct”.

Ces erreurs de segmentation, souvent incompréhensibles au premier abord, ont été l’une des principales causes de découragement au cours du projet. Mais je n’ai pas baissé les bras. À chaque « segfault », j’ai pris le temps de déboguer le programme, en plaçant des printf à des endroits stratégiques afin de comprendre où le code s’arrêtait et pourquoi.

J’ai alors vérifié, étape par étape, toutes les allocations mémoire, les appels de fonctions et surtout l’ordre d’utilisation des arguments. J’ai mis en place de nombreux contrôles internes pour m’assurer que chaque variable, chaque table et chaque pointeur étaient correctement alloués, initialisés et libérés.

Ce travail de vérification minutieuse, accompagné des nombreux tests ajoutés dans les fichiers *control_t3c.c* et *control_dict.c*, m’a finalement permis de corriger les erreurs de segmentation et surtout de comprendre l’importance cruciale de la gestion mémoire en C.

Cette expérience, parfois éprouvante, m’a appris à ne jamais négliger la vérification des entrées et sorties à chaque étape du programme, et à considérer chaque bug comme une opportunité de progresser dans la compréhension du langage.

VII. Conclusion

Rain-C démontre qu’il est possible d’écrire, en C, un outil de génération et de recherche de rainbow tables à la fois performant, modulaire et portable. L’organisation du code en modules distincts (*hash*, *T3C*, *dict*, *menu*) assure une clarté fonctionnelle et une séparation nette des responsabilités entre lecture, calcul, recherche et interface.

Les choix techniques, comme le recours à OpenSSL pour les fonctions de hachage, utilisation d’un arbre binaire pour l’indexation et architecture simple en mémoire, garantissent un outil efficace et reproductible.