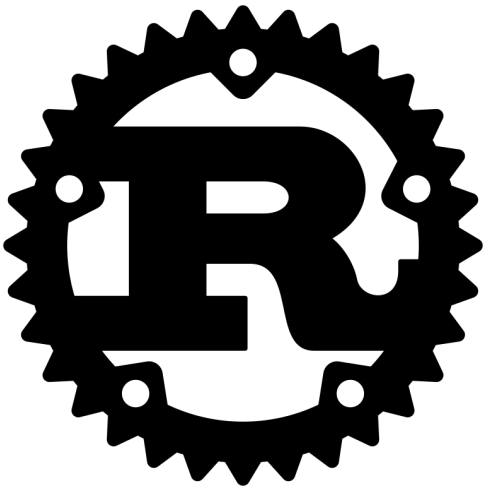
Rapport de soutenance - Projet rPack

ByteShield

26/02/2025

Louis BEAUCAMPS
Arthur WAMBST
Marcus POIROT
Gabriel TONGBOONNAK

Contents



1 Introduction

1.1 Qu'est ce qu'un packer ?

Un **packer** est un outil d'encapsulation d'exécutables. Il sert à complexifier l'analyse statique et dynamique d'un binaire. L'objectif est de protéger le programme tout en garantissant son exécution normale et en limitant l'impact sur les performances de celui-ci. Cela passe par plusieurs procédés tel que la compression et le chiffrement de celui-ci, ainsi que l'implémentation de plusieurs méthode d'anti-débogage.

1.2 Etat de l'art

De très nombreux projets de packer existent et sont facilement trouvable en ligne (https://github.com/packing-box/awesome-executable-packing).

Exemple de UPX :

UPX (Ultimate Packer for eXecutables) est l'un des packers les plus connus. Il est utilisé pour compresser les fichiers exécutables sur plusieurs systèmes d'exploitation.

Il fonctionne de la manière suivante:

- L'exécutable est analysé pour identifier les sections (code, données, etc.). Ces sections sont compressées avec un algorithme comme LZ77.
- Un petit programme stub est ajouté à l'exécutable compressé. Ce stub est chargé de décompresser les données en mémoire lors de l'exécution.

- Lorsqu'on exécute l'exécutable packé, le stub charge le fichier compressé en mémoire, décompresse les données, puis exécute le code d'origine.

Après UPX, Themida est un autre packer très connu, mais il est utilisé principalement pour protéger les logiciels commerciaux.

Contrairement à UPX, qui est simple, gratuit et qui a pour objectif de compresser des données, Themida offre des fonctionnalités avancées comme:

- l'obfuscation
- la détection de débogueurs et d'outils de virtualisation
- la virtualisation du code

VMProtect, un packer principalement utilisé pour protéger des logiciels données critiques.

Ensuite, VMProtect est un packer avancé utilisé principalement pour protéger des logiciels contre la rétro-ingénierie, le piratage et le cracking.

La différence avec Themida est que VMProtect est utilisé pour des logiciels nécessitant une protection maximale, comme des programmes critiques ou des algorithmes confidentiels.

1.3 Notre projet

Nous avons pour ambition de développer un packer pour des exécutables ELF. Cette idée s'inscrit dans notre intérêt pour la cybersécurité, et plus particulièrement pour la rétro-ingénierie. C'est cette passion qui nous a motivés à concevoir un tel outil. Le choix du langage Rust s'avère particulièrement pertinent pour ce projet. En effet, Rust complique considérablement les tentatives de rétro-ingénierie grâce à son compilateur, qui produit un code machine optimisé et difficile à analyser. Cela s'explique notamment par des techniques telles que l'inlining agressif, la suppression des symboles inutiles, et la génération de noms et structures complexes. En somme, un packer écrit en Rust offre une protection robuste contre la rétro-ingénierie tout en maintenant de bonnes performances pour l'exécutable.

1.4 L'équipe

Nous sommes un groupe de 4 étudiants d'EPITA :

Louis BEAUCAMPS (chef de projet)
Arthur WAMBST
Marcus POIROT
Gabriel TONGBOONNAK

2 Avancée du Projet

2.1 Introduction

Actuellement, nous avons implémenté les fonctionnalités suivantes, bien que seules la compression et l'AES soient intégrées a l'exécutable en sortie :

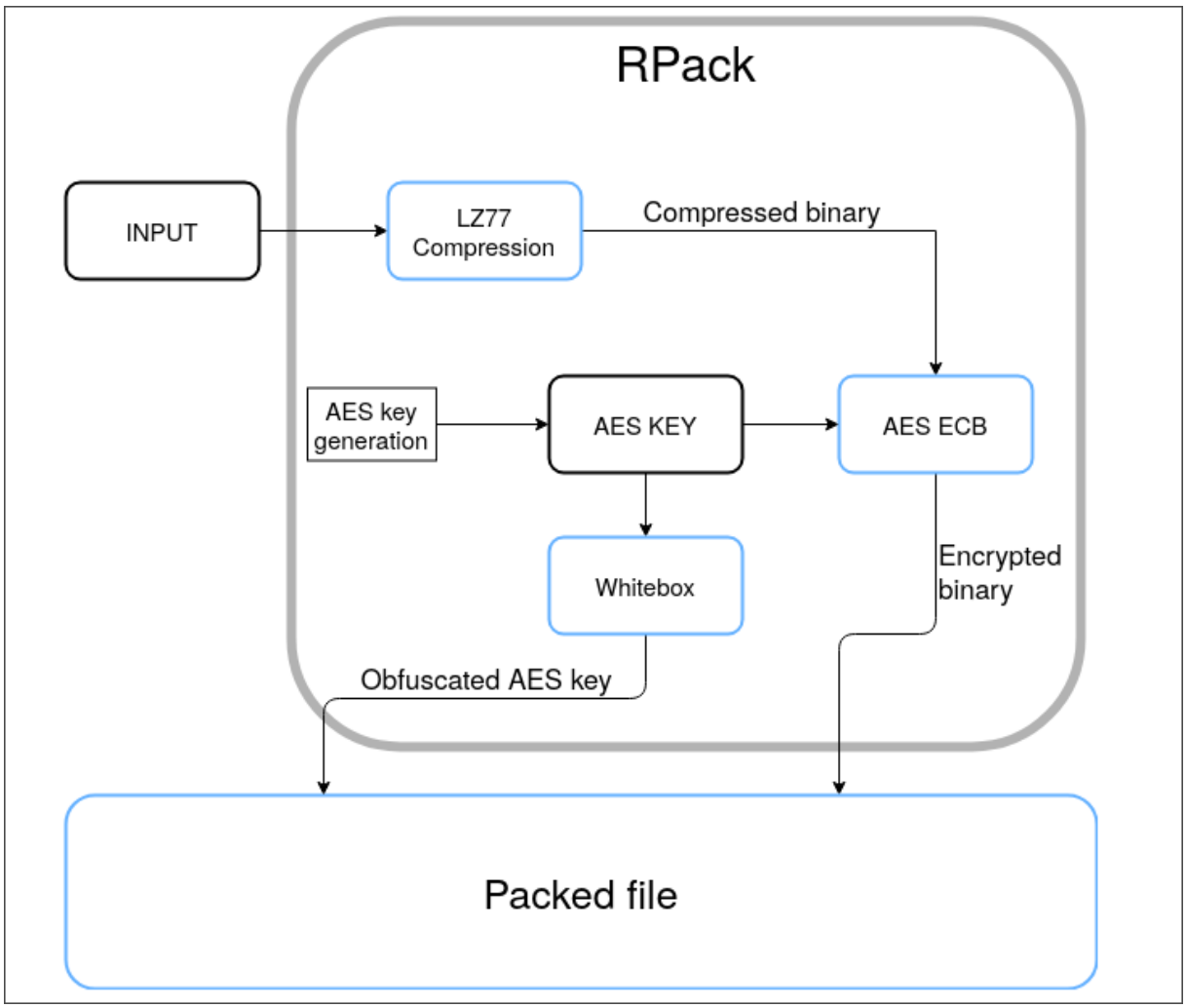


Figure 1: Packing process

Nous n'avons donc pas eu spécialement de problèmes majeurs dans la partie statique, en revanche la partie dynamique nous a forcé à fortement adapter le projet.

En effet, nous voulons que le fichier final packé ressemble a ceci, qui sera complété pour les prochaines soutenances par d'autres mecanismes :

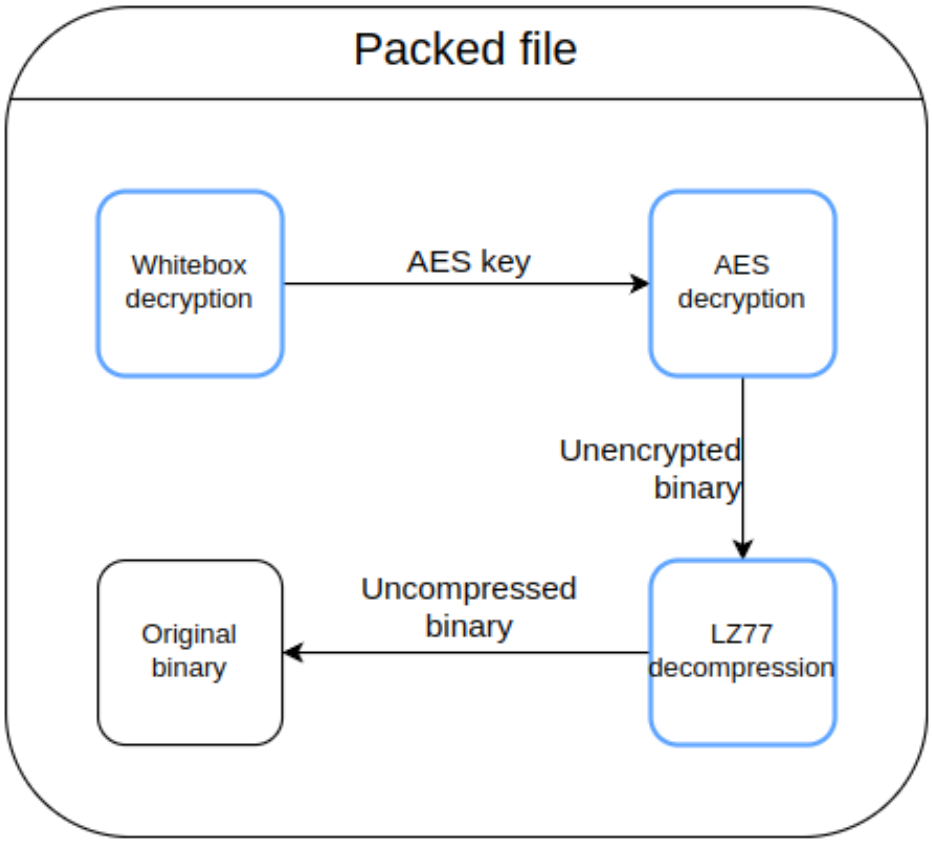


Figure 2: Packing process

Mais nous avons rencontré beaucoup plus de difficultés qu'attendu, surtout car en rust, il n'existe pas de fonction permettant de lancer l'exécutable et de lire le code assembleur qu'il exécute. Ceci nous a coûte beaucoup de temps, et nous a force a nous adapte en utilisant des crates, détaillées dans la partie '3 Problèmes rencontres =¿ 3.1 Exécution en mémoire'

2.2 White Box

Qu'est-ce qu'une White Box ? La White Box est une méthode conçue pour protéger les clés cryptographiques dans des environnements non sécurisés où l'attaquant a un accès complet au système (dans notre cas l'exécutable), cela inclue la mémoire et le code exécuté. Contrairement à la cryptographie traditionnelle, où la clé est supposée rester secrète, une White Box intègre la clé directement dans les algorithmes de chiffrement, rendant son extraction particulièrement difficile. Ce mécanisme vise à résister à diverses attaques, notamment l'analyse statique, dynamique ou les attaques par canaux auxiliaires (side-channel attacks).

Utilisation des Lattices

Les lattices (réseau géométrique) constituent une structure mathématique composée de points disposés selon un motif répétitif.

Bien que simples en apparence, ces structures sous-tendent des problèmes mathématiques extrêmement complexes, ce qui en fait une base prometteuse pour la cryptographie post-quantique. Avec l'avènement des ordinateurs quantiques, de nombreux systèmes cryptographiques actuels, comme RSA ou ECC, deviendront vulnérables.
La cryptographie post-quantique vise donc à développer des algorithmes résistants aux attaques quantiques, parmi lesquels ceux basés sur les lattices sont particulièrement prometteurs.

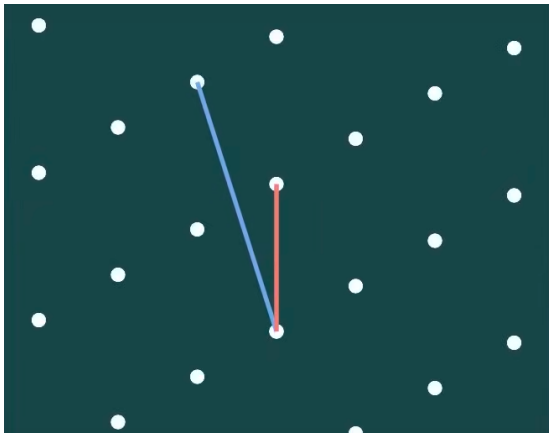


Figure 3: Exemple de lattice de 2 dimensions

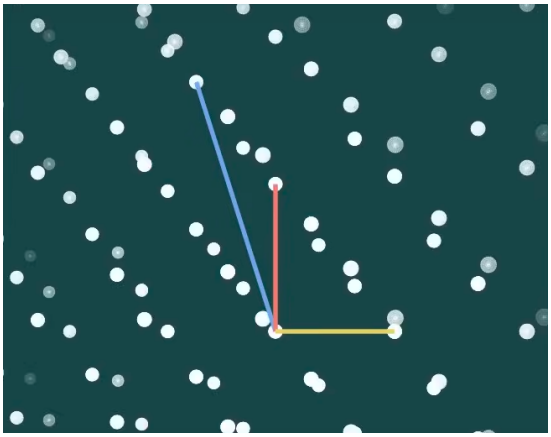


Figure 4: Exemple de lattice de 3 dimensions

Un problème fondamental associé aux lattices est le *Shortest Vector Problem* (SVP), qui consiste à trouver le point du lattice le plus proche de l'origine (hormis l'origine elle-même).

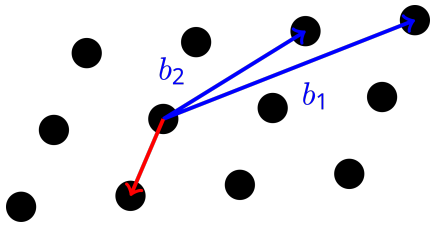


Figure 5: Exemple du problème SVP

Une autre variante, le *Closest Vector Problem* (CVP), vise à déterminer le point du lattice le plus proche d'un point donné dans l'espace.

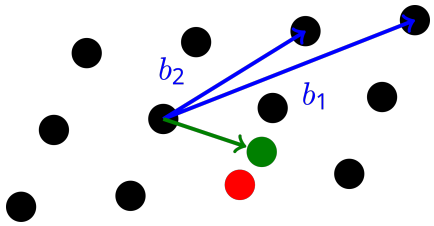


Figure 6: Exemple du problème CVP

Ces problèmes deviennent exponentiellement plus complexes en haute dimension, rendant leur résolution extrêmement difficile, même pour un ordinateur quantique. Cette propriété les rend intéressants pour la cryptographie post-quantique.

Dans ce contexte, l'un des schémas cryptographiques classiques utilisant les lattices est le chiffrement de GGH (Goldreich-Goldwasser-Halevi).

Son principe repose sur l'existence de deux bases pour un même lattice : une *bonne* base (avec des vecteurs presque orthogonaux), qui permet de résoudre facilement le CVP, et une *mauvaise* base (avec des vecteurs quasi alignés), rendant la résolution du CVP très difficile.

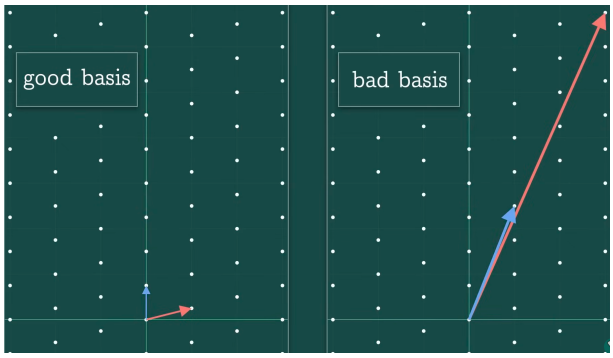


Figure 7: Exemple de bonne et mauvaises bases

Cependant, le schéma GGH n'est pas sécurisé en pratique, car il peut être contourné par certaines attaques mathématiques.

Néanmoins, plusieurs algorithmes cryptographiques modernes exploitent toujours les lattices pour assurer une sécurité renforcée contre les attaques quantiques.

Parmi eux, Kyber, Dilithium et Falcon font partie des standards retenus pour la cryptographie post-quantique.
Une approche clé repose sur le problème *Learning With Errors* (LWE), qui introduit des erreurs dans les calculs de manière contrôlée afin de complexifier les attaques.

Lien avec notre implémentation White Box

Dans le cadre de notre projet, nous cherchons à protéger la clé AES en l'intégrant dans une White Box.
La conception d'une implémentation White Box véritablement efficace requiert une expertise mathématique approfondie, que nous ne possédons pas encore. C'est pourquoi, plutôt que de développer une White Box potentiellement vulnérable en partant de zéro, nous avons privilégié la réimplémentation d'une solution existante. Dans un souci de renforcer la sécurité, nous avons choisi de nous orienter vers un algorithme moins répandu. L'objectif est de limiter l'exposition aux attaques courantes et aux écosystèmes d'attaques ayant déjà été largement explorés par la communauté des chercheurs en sécurité. Notre choix s'est donc porté sur l'implémentation d'une White Box asymétrique basée sur la cryptographie sur réseaux euclidiens. Les lattices offrent une alternative séduisante aux systèmes de clés publiques traditionnels tels que RSA et ECC, notamment en raison de leur robustesse reconnue et d'un nombre comparativement plus faible d'attaques connues à ce jour. Nos recherches nous ont menés au projet open-source "BVWhiteBox" (https://github.com/quarkslab/BVWhiteBox), qui fournit une implémentation white-box basée sur les réseaux euclidiens. Nous avons décidé de nous appuyer sur cette base, en la réimplémentant en Rust afin d'étudier et d'adapter davantage cette technique prometteuse. L'implémentation résultante génère des lattices de dimension 512, ce qui confère un niveau de sécurité élevé.

2.2.1 Fondements mathématiques : Ring-LWE (Learning With Errors)

Le problème *Ring Learning With Errors* (RLWE) constitue la base théorique de notre implémentation.

Soit : - Un anneau polynomial $R_q = Z_q[X]/(X^n + 1)$ où q est premier et $n = 2^k$ - Une distribution de bruit χ (gaussienne discrète) - Une clé secrète $s \in R_q$

Le problème RLWE consiste à distinguer entre : 1. Des paires $(a_i, b_i = a_i \cdot s + e_i)$ où $a_i \leftarrow \$R_q$, $e_i \leftarrow \chi$ 2. Des paires (a_i, b_i) totalement aléatoires

La sécurité repose sur l'impossibilité pratique de séparer le bruit e_i de la structure algébrique, même avec un ordinateur quantique.

2.2.2 Transformation White Box par décomposition RNS/NTT

Pour masquer la clé secrète s, le déchiffrement $m = c_2 - c_1 \cdot s$ est décomposé via :

- **Number Theoretic Transform (NTT)** : Représente les polynômes en évaluations sur des racines de l'unité ω, convertissant les produits polynomiaux en produits coefficients par coefficient.
- **Residue Number System (RNS)** : Décompose les coefficients en résidus modulo une base $\{p_1, ..., p_k\}$, réduisant les calculs à des entiers de 4-8 bits.

2.2.3 Encodages homomorphes et masquage

Trois couches de protection sont intégrées :

- **Zéro algébrique** : $S_{sk}[a, b] = (a - b \cdot sk) + \mathsf{Enc}(0)$, neutre après déchiffrement
- **Masque multiplicatif** : $\mathsf{Enc}(X^k)$ pour permuter les coefficients
- **Encodages externes** : Fonctions non linéaires G, H appliquées avant chiffrement, nécessitant un tiers de confiance

2.2.4 Génération des tables de déchiffrement

Le processus clé comprend :

1. Décomposition RNS/NTT de la clé secrète s et des paramètres
2. Précalcul des résidus $Q = (a \cdot s + e)\%p_i$ via l'algorithme de Montgomery
3. Stockage dans trois jeux de tables :
   - Tables V/W : Calculs linéaires masqués (32-64 Mo)
   - Table T : Réduction modulo 2 non linéaire (200 Mo)

2.2.5 Paramètres techniques et choix critiques

Notre implémentation spécifie des paramètres optimisés pour équilibrer sécurité et performances :

- **Dimension du réseau** : $n = 512$ - Garantit une sécurité post-quantique ¿=128 bits contre les attaques par réduction de réseau (BKZ-100)
- **Module premier** : $q = 1\,231\,873$ - De la forme $q = 1024i + 1$ (avec $i = 1203$), permettant une NTT efficace grâce à l'existence de racines $2n$-ièmes de l'unité
- **Bases RNS** :
  - $\beta = [13, 16, 19, 27, 29]$ (produit 3 094 416)
  - $\beta' = [11, 17, 23, 25, 31]$ (produit 3 333 275)

  Choix motivé par : - Couverture conjointe $\beta \times \beta' > q^2$ pour les calculs Montgomery - Optimisation des opérations modulaires sur 4-8 bits
- **Niveau de défi (chal=2)** : Active deux couches d'encodages homomorphes (zéro + rotation), avec : - Masquage par X^{256} dans R_q - Erreurs Gaussiennes $\sigma = 3.2$ pour le RLWE

2.2.6 Sécurité des choix paramétriques

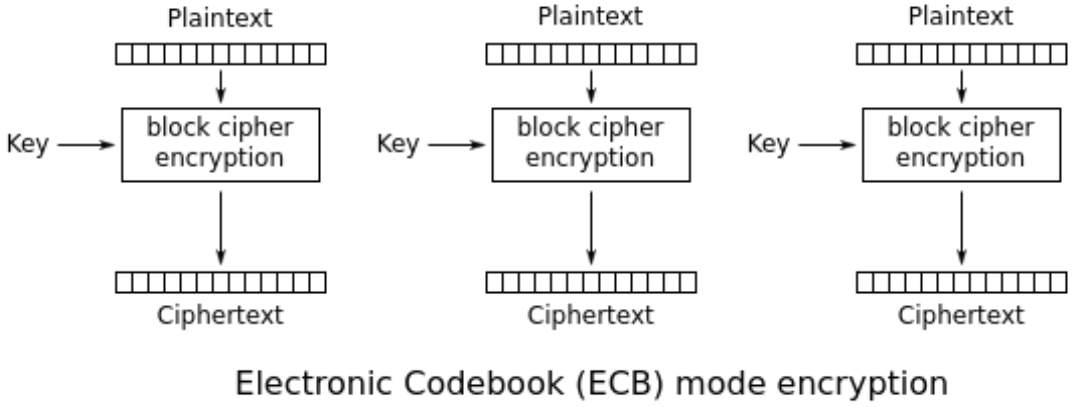
Paramètre	Robustesse
Dimension $n = 512$	Résiste à BKZ-100 (+- =100 ans de calcul quantique)
Module $q \approx 2^{20}$	Évite les attaques par arrondi/erreur
Bases RNS mutuellement premières	Empêche les corrélations inter-modulaires
Taille des résidus 4-8 bits	Limite les attaques par canaux auxiliaires

Justification théorique : La configuration (n, q, σ) suit les recommandations du NIST PQC Round 3 pour Kyber-512, avec une marge de sécurité supplémentaire via l'augmentation de q.

Cette implémentation de la whitebox bien que complexe garantis une bonne sécurité de la clé aes. Si jamais vous voulez plus d'informations sur la white box n'hésitez pas à consulter le document de thèse de Lucas Barthélémy: https://github.com/quarkslab/BVWhiteBox/blob/master/t

2.3 AES

Chiffrement AES-128 ECB : Le chiffrement AES-128 ECB est un mode de chiffrement rapide et robuste pour le traitement de fichiers, car il traite chaque bloc de données indépendamment tout en utilisant une clé symétrique de 128 bits. Cette clé assure un niveau élevé de sécurité en rendant difficile toute tentative de bruteforce, tout en offrant une excellente rapidité et efficacité grâce à sa simplicité de mise en œuvre. Cela le rend particulièrement adapté à notre situation, nécessitant un bon compromis entre performance et sécurité.
De plus, nous avons choisi d'utiliser une clé de 128 bits plutôt que de 256 bits afin d'optimiser les performances.



Electronic Codebook (ECB) mode encryption

Génération de la clé : Dans un premier temps, nous devons générer une clé secrète . Cette clé de 16 bytes (soit 128 bits) servira de fondation pour les étapes de chiffrement et de déchiffrement. Cette génération doit être effectuée de manière sécurisée afin de garantir la confidentialité des données. Puis une fois générée, elle doit être stockée de façon sécurisée et ne jamais être exposée dans un environnement non sécurisé.

Chiffrement : Pour chiffrer nos données, nous les divisons en blocs de 16 bytes. Cette segmentation permet d'optimiser le traitement et garantit une compatibilité avec la taille de bloc imposée par l'algorithme AES-128. Ensuite chaque bloc est chiffré individuellement.

Déchiffrement : Avant l'exécution, nous devons déchiffrer les données chiffrées afin de retrouver le contenu original.
Ce processus est simplement l'inverse du chiffrement. Chaque bloc chiffré de 16 bytes est déchiffrer en utilisant la même clé secrète. Une fois tous les blocs déchiffrés, les données originales sont reconstituées et prêtes à être utilisées.

2.4 LZ77

Nous avons étudié l'algorithme de compression LZ77 dans le cadre de notre projet en Rust, afin d'optimiser le stockage et la transmission de données.

Introduction La compression de données est un enjeu majeur dans le stockage et la transmission d'informations. L'algorithme LZ77, grâce à son utilisation d'une fenêtre glissante, permet de réduire efficacement la taille des fichiers sans perte d'information.

Compression avec Rust L'implémentation de LZ77 en Rust tire parti de la gestion fine de la mémoire et des fonctionnalités avancées du langage pour améliorer la rapidité et l'efficacité de la compression. L'utilisation des slices et des structures de données optimisées permet une gestion efficace de la fenêtre glissante, élément clé de l'algorithme.

L'algorithme fonctionne en explorant une séquence de données et en identifiant les parties déjà apparues dans la fenêtre glissante. Cette approche permet de réduire la taille des fichiers en remplaçant les motifs répétés par des références aux segments précédents.

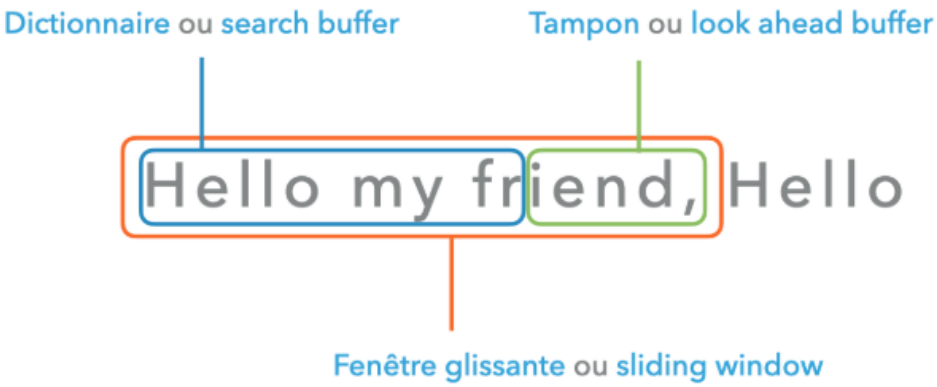


Figure représentant une séquence de données sous forme d'une chaîne de caractères.

Implémentation Nous avons structuré notre implémentation en Rust en plusieurs étapes :

- Utilisation d'un buffer circulaire pour gérer efficacement la fenêtre glissante.
- Recherche des répétitions via une approche optimisée en mémoire.
- Encodage des triplets (distance, longueur, caractère) pour représenter les motifs compressés.

L'intégration de Rust dans ce projet nous a permis d'exploiter ses fonctionnalités de gestion de la mémoire sans coût de garbage collector, améliorant ainsi la performance et réduisant la latence.

Avantages et défis de Rust Rust nous a offert plusieurs avantages pour cette implémentation :

- Sécurité mémoire grâce au système de propriété et d'emprunt.
- Gestion efficace de la mémoire sans besoin de garbage collector.
- Performances accrues pour le traitement de grandes quantités de données.

Cependant, quelques défis ont dû être surmontés :

- Complexité de la gestion des références et des emprunts.
- Temps d'apprentissage du langage pour optimiser l'implémentation.
- Ajustement des structures de données pour garantir des performances optimales.

Résultats et perspectives Des tests montrent que des implémentations en Rust offre un gain de performance notable par rapport à d'autres langages comme Python ou Java, en raison de son approche bas niveau et de l'absence de collecte de déchets automatique.

Dans le futur, des optimisations supplémentaires pourraient être explorées si besoin, telles que l'utilisation du codage de Huffman et l'implémentation de variantes de LZ77 adaptées aux flux de données en temps réel. Une amélioration du multithreading pourrait également être envisagée pour tirer parti des architectures modernes.

Décompression La décompression des données compressées avec LZ77 suit un processus inverse de l'encodage, permettant de restaurer fidèlement le fichier original.

- **Lecture des données** : Les triplets encodés sont analysés afin d'extraire les informations sur la position des motifs et leur longueur.
- **Reconstruction des motifs répétés** : L'algorithme utilise la fenêtre glissante pour recopier les séquences déjà présentes et reconstruire progressivement les données.
- **Reconstruction complète des données** : Le processus se poursuit jusqu'à la restitution complète du fichier initial, garantissant une parfaite fidélité aux données originales.

LZ77 est un algorithme fondamental utilisé dans de nombreux formats de compression comme ZIP et PNG, grâce à son efficacité dans le traitement des motifs répétitifs. Son approche permet une récupération rapide et fidèle des données, le rendant incontournable dans les solutions modernes de compression.

3 Problèmes rencontrés

3.1 Exécution en mémoire du guest

Nous pensions naïvement qu'il y aurai une méthode rust pour pouvoir exécuter un exécutable et qu'il faudrait alors juste lire l'assembleur puis exécuter les instructions, ce qui est indispensable afin de pouvoir implémenter toutes les étapes dynamiques de notre projet. Malheureusement, même après de nombreuses recherches, nous n'avons pas trouve ce que nous cherchions.

Finalement, nous avons trouvé cette série d'articles, qui nous évitent de tout devoir réinventer de zéro:
https://fasterthanli.me/series/making-our-own-executable-packer

Voici un résume des principales étapes que nous avons suivi :

Son packer est divisé en plusieurs crates:

Pixie : définit le format de fichier Pixie. Il encapsule le binaire et les fichiers nécessaires (manifeste) dans ce format. Il structure le fichier Pixie pour que les loaders (stage1, stage2) puissent facilement extraire et exécuter le programme, et accéder aux fichiers embarqués via le VFS. En bref, pixie crée le fichier Pixie packagé à partir de votre binaire et de sa configuration.

Encore : environnement d'exécution minimaliste (runtime) au sein duquel les exécutables Pixie s'exécutent. Il gère les interactions bas niveau avec le système, la mémoire et le système de fichiers virtuel (VFS).

Minipak : Outil en ligne de commandes qui génère le fichier packé. C'est le point d'entrée où l'on va ajouter nos étapes de compressions et de chiffrement.

Stage 1 : Contient le code de démarrage initial et minimal du Pixie. C'est la première phase du processus de lancement, très bas niveau.

Stage 2 : Représente la seconde phase du démarrage, plus élaborée que stage1. stage2 initialise l'environnement encore, charge le programme principal et configure le VFS avant de lancer l'exécution du programme empaqueté. C'est ici que l'on ajoute nos étapes de déchiffrement et de décompression.

Voici ensuite des explications en détail du fonctionnement et de notre utilisation de ces crates :

3.1.1 Pixie

La crate pixie définis et implémente le format de fichier Pixie.

Ce format custom, a pour objectif d'encapsuler un elf.
Il orchestre la création des fichiers Pixie en intégrant le binaire principal et les ressources additionnelles définies dans le manifeste, et structure le fichier résultant en segments, incluant un en-tête dédié, afin de faciliter le démarrage et l'exécution du programme packé.

Cette structuration permet aux phases de lancement (stage1, stage2) d'extraire efficacement le binaire et de gérer l'accès aux ressources embarquées via un système de fichiers virtuel (VFS) lors de l'exécution.

En résumé, pixie assure la transformation du binaire en entrée vers un format d'exécutable personnalisé, en définissant un format adapté et en fournissant les mécanismes nécessaires pour pouvoir l'exécuter dans les autres crates.

3.1.2 Encore

La crate encore est une librairie de bas niveau essentielle au bon fonctionnement de l'environnement d'exécution Pixie. Son rôle principal est de fournir une **abstraction sûre et utilisable des appels système (syscalls)** pour l'environnement d'exécution Pixie. Il se positionne comme une couche d'interface entre le code Rust de haut niveau et le noyau du système d'exploitation sous-jacent, en particulier pour les opérations fondamentales telles que la gestion de la mémoire, les entrées/sorties et le contrôle du processus.

La crate encapsule de manière sécurisée les **appels système Linux standards** (tels que ‘mmap‘, ‘munmap‘, ‘open‘, ‘write‘, ‘exit‘, etc.) en utilisant de l'assembleur inline. Ce module définit des structures Rust et des fonctions qui permettent d'invoquer ces syscalls de manière contrôlée et portable, en masquant la complexité et les aspects potentiellement dangereux de l'interaction directe avec le noyau. encore fournit ainsi une **API** pour effectuer des opérations système essentielles.

En complément de l'abstraction des syscalls, le crate offre des **utilitaires de plus haut niveau** qui s'appuient sur ces primitives. Il fournit des abstractions pour la manipulation de fichiers (ouverture, lecture, écriture, mapping mémoire), simplifie la gestion de la mémoire virtuelle via le syscall `mmap`. Il permet également d'accéder à l'environnement du processus, notamment aux arguments en ligne de commande et aux vecteurs auxiliaires. Enfin, il contient des éléments fondamentaux comme le gestionnaire de panique et l'initialisation de l'allocateur mémoire global, indispensables pour un environnement d'exécution minimaliste.

En résumé, le crate `encore` est une **librairie système de base** qui fournit les fondations nécessaires à l'exécution des programmes Pixie. Il offre une interface Rust sûre et abstraite pour interagir avec le système d'exploitation, en gérant les opérations de bas niveau critiques telles que les appels système, la gestion de la mémoire et les opérations de fichiers.

3.1.3 Stage1

Le crate **stage1** représente la première phase d'initialisation lors du démarrage d'un exécutable Pixie. Il s'agit d'un composant **minimaliste et autonome** dont le rôle principal est d'amorcer l'environnement d'exécution et de transférer le contrôle à la phase suivante, `stage2`. Il est conçu pour être le **point d'entrée initial** du Pixie, exécuté directement par le système d'exploitation.

Sa première tâche est de mettre en place un **environnement d'exécution rudimentaire**. Cela inclut notamment l'initialisation de l'allocateur mémoire, permettant ainsi d'allouer dynamiquement de la mémoire pour les étapes suivantes. 'stage1' procède ensuite à l'**analyse de son propre fichier exécutable Pixie**. En s'ouvrant lui-même via le système de fichiers, il accède au manifeste Pixie embarqué, qui contient des informations cruciales sur la structure du fichier et les composants qu'il contient.

Grâce au manifeste, il est capable de localiser et de **charger en mémoire le composant** `stage2` depuis le fichier Pixie. Il utilise les fonctionnalités du crate **pixie** pour interpréter et manipuler les structures ELF de `stage2`, et celles du crate **encore** pour effectuer les opérations de mapping mémoire nécessaires. Une étape essentielle est la **relocation de** `stage2` **en mémoire**. Cette opération garantit que le code de `stage2` pourra s'exécuter correctement à l'adresse où il a été chargé, une pratique courante pour les exécutables relogeables. Enfin, il **transfère l'exécution à** `stage2` en sautant vers son point d'entrée, achevant ainsi sa mission d'amorçage et passant le relais à la phase suivante du processus de démarrage Pixie.

En résumé, `stage1` est un **bootstrap loader** minimal qui assure la transition initiale vers l'environnement d'exécution Pixie. Il prépare le terrain pour `stage2` en initialisant l'environnement de base, en chargeant `stage2` en mémoire et en lui cédant le contrôle, permettant ainsi de poursuivre le processus de démarrage plus complexe du Pixie.

3.1.4 Stage2

Le crate **stage2** constitue la seconde phase du processus de démarrage d'un exécutable Pixie, succédant à `stage1`.

Son rôle consiste à **préparer l'environnement d'exécution final pour le programme embarqué**, désigné ici comme le "guest".
Il prend le relais de `stage1` et orchestre les étapes nécessaires pour que le programme guest puisse être lancé et s'exécuter correctement au sein de l'environnement Pixie.

La première tâche de `stage2` est de procéder à la **sécurisation et à la décompression du programme guest**. Il décompresse en utilisant l'algorithme LZ4. Ces étapes visent à protéger le programme embarqué et à optimiser la taille du fichier Pixie. Une fois le guest décompressé en mémoire.

Ensuite, `stage2` prend en charge le **mapping en mémoire du guest**. Il détermine l'adresse de base de chargement en fonction du caractère relogeable ou non du guest, et utilise les fonctionnalités de mapping mémoire du crate **encore** pour charger le guest à l'adresse appropriée. Une fois le guest mappé, `stage2` **configure l'environnement d'exécution** en ajustant les vecteurs auxiliaires (auxv), notamment `AT_PHDR`, `AT_PHNUM` et `AT_ENTRY`, pour qu'ils pointent vers les segments de programme et le point d'entrée du guest.

Enfin, `stage2` gère optionnellement le cas où le guest requiert un **interpréteur (dynamic linker)**.
Si un segment `Interp` est présent dans le guest, `stage2` charge l'interpréteur spécifié, le mappe en mémoire, et ajuste le vecteur auxiliaire `AT_BASE` pour indiquer son adresse de base.

Dans tous les cas, avec ou sans interpréteur, `stage2` termine son exécution en **transférant le contrôle au point d'entrée du guest** via une fonction de lancement fournie par le crate **pixie**, permettant ainsi le démarrage effectif du programme embarqué.

En résumé, `stage2` est la phase de démarrage avancée du Pixie, responsable du déchiffrement, de la décompression, du chargement en mémoire, de la configuration de l'environnement et du lancement du programme guest.
Il assure la transition entre l'initialisation minimale de `stage1` et l'exécution du programme applicatif embarqué, en gérant des aspects complexes comme le chargement ELF, la relocation et la gestion de l'interpréteur.

3.1.5 minipak

Le crate **minipak** est responsable de la **création des exécutables Pixie**.

Il s'agit d'un utilitaire en ligne de commande qui prend en entrée un exécutable ELF standard ("guest") et le transforme en un exécutable Pixie autonome, prêt à être exécuté par l'environnement d'exécution Pixie.

Il orchestre l'ensemble du processus d'empaquetage, intégrant les composants `stage1` et `stage2`, compressant le programme "guest", et générant le manifeste Pixie nécessaire au démarrage.

Le processus d'empaquetage réalisé par `minipak` peut être décomposé en plusieurs étapes clés :

1. **Compilation et intégration de `stage1` et `stage2`** : Lors de la phase de build, `minipak` compile les crates `stage1` et `stage2` en tant que librairies statiques (**cdylib**). Ces librairies sont ensuite intégrées à l'exécutable `minipak` lui-même, prêtes à être embarquées dans le futur exécutable Pixie.
2. **Analyse et traitement de l'exécutable "guest"** : `minipak` prend en entrée le chemin vers un exécutable ELF ("guest") spécifié par l'utilisateur via la ligne de commande. Il analyse cet exécutable, notamment sa structure ELF et ses segments, en utilisant les fonctionnalités du crate `pixie`.
3. **Relocation de `stage1`** : `minipak` effectue une relocation de `stage1` en mémoire virtuelle, en choisissant une adresse de base appropriée. Cette étape prépare `stage1` à être inclus dans l'exécutable Pixie final.
4. **Intégration de `stage2`** : Le code compilé de `stage2` est intégré tel quel dans la section de données de l'exécutable `minipak`, à un offset spécifique.
5. **Compression du "guest"** : Pour optimiser la taille de l'exécutable Pixie et protéger le programme embarqué, `minipak` compresse le "guest" en utilisant l'algorithme LZ4. Ces étapes réduisent l'empreinte du Pixie et ajoutent une couche de sécurité.
6. **Génération du manifeste Pixie** : `minipak` crée un manifeste Pixie qui décrit la structure de l'exécutable final, notamment les offsets et tailles de `stage2` et du "guest" chiffré et compressé. Ce manifeste est essentiel pour que `stage1` et `stage2` puissent localiser et charger correctement les différents composants au démarrage.
7. **Ecriture de l'exécutable Pixie final** : Enfin, `minipak` assemble tous les éléments (header ELF, `stage1` relogé, `stage2`, "guest" chiffré et compressé, manifeste Pixie, marqueur de fin) dans un nouveau fichier exécutable, qui constitue l'exécutable Pixie final.

En résumé, `minipak` est l'outil d'empaquetage central du projet. Il automatise la création d'exécutables Pixie autonomes et sécurisés en intégrant les phases d'initialisation `stage1` et `stage2`, en compressant et chiffrant le programme embarqué, et en générant les métadonnées nécessaires au démarrage. `minipak` est donc un composant clé pour la distribution et l'exécution des programmes Pixie.

3.2 Contraintes liés à Rust pour le Développement Bas Niveau

Le choix du langage **Rust** pour rPack a été motivé par ses qualités en termes de **sécurité mémoire, de performance et de sûreté**, et par le fait que certaines de ses caractéristiques, comme la monomorphisation, peuvent rendre la rétro-ingénierie des binaires produits plus ardue.

Rust est réputé pour son système de typage rigoureux, son ownership model et son borrow checker, qui permettent de prévenir de nombreuses erreurs de programmation, en particulier celles liées à la gestion de la mémoire. Cependant, notre projet nécessite de manipuler la mémoire et de produire du code bas niveau, ce qui confronte Rust à ses limites et impose des compromis, notamment en ce qui concerne la sécurité et l'utilisation du code `unsafe` qui est utilisé sur la plupart des fonctions.

En effet, l'essence même du packer réside dans la **manipulation directe de la mémoire et des appels système**. Les crates `encore`, `stage1` et `stage2` interagissent intimement avec le kernel pour réaliser des opérations fondamentales : allocation et gestion de la mémoire virtuelle (`mmap`, `munmap`), lecture et écriture de fichiers, exécution de code arbitraire en mémoire, etc. Ces opérations, par nature, **échappent au modèle de sécurité de Rust** et nécessitent l'utilisation de blocs de code `unsafe`.

L'utilisation de code `unsafe` est ainsi **omniprésente** dans les crates de bas niveau du projet.

On la retrouve notamment pour :

- **Les points d'entrée (`entry`, `_start`)** : Ces fonctions, qui constituent les points de départ de l'exécution, sont nécessairement déclarées comme `unsafe` et implémentées en assembleur inline pour interagir directement avec le chargeur et le système d'exploitation au niveau le plus bas.
- **Les appels système (syscalls)** : Le crate `encore` encapsule les appels système Linux standards en utilisant de l'assembleur inline et des fonctions `unsafe`, car ces interactions directes avec le noyau sont intrinsèquement non sûres.
- **Le mapping mémoire (mmap)** : La gestion de la mémoire virtuelle, cruciale pour le chargement et la relocation des exécutables, repose sur l'appel système `mmap` et implique donc l'utilisation de code `unsafe` pour manipuler directement les adresses mémoire.
- **L'analyse et la manipulation des structures ELF** : Le crate `pixie` doit analyser et manipuler des structures de données binaires complexes (headers ELF, segments, tables de symboles), ce qui requiert souvent l'interprétation de données brutes en mémoire et l'utilisation de raw pointers, nécessitant des blocs `unsafe`.

Face à cette omniprésence du code `unsafe`, l'approche adoptée dans rPack a été de **contenir et de maîtriser au maximum son utilisation**.

L'objectif n'est pas de supprimer le code `unsafe` (ce qui serait impossible pour un projet de cette nature), mais de le **restreindre à des modules spécifiques et bien isolés** (comme `encore` et certaines parties de `pixie`) et de construire des **abstractions sûres** au-dessus de ces primitives non sûres.
L'API fournie par le crate `encore`, par exemple, vise à offrir une interface Rust conviviale et sécurisée pour les opérations système, même si son implémentation interne repose sur du code `unsafe`.

En conclusion, le développement de rPack en Rust représente un **compromis constant entre la sécurité et la nécessité d'opérations bas niveau**.

Si le langage Rust apporte des garanties précieuses en termes de sûreté et de performance, la nature même du projet impose l'utilisation de code `unsafe` pour interagir avec le système d'exploitation et manipuler directement la mémoire.
L'effort principal réside alors dans la **gestion rigoureuse** de ce code `unsafe`, en le confinant et en construisant des abstractions sûres pour minimiser les risques et préserver autant que possible les avantages de Rust.

Globalement, le Rust est un choix pertinent pour les applications *userland* grâce à ses garanties de sécurité et de performance. Cependant, le développement bas niveau nécessitant de manipuler la mémoire, dont notre *packer*, nécessite un usage important de code `unsafe`, ce qui, bien que nécessaire dans ce contexte, va à l'encontre des principes fondamentaux du language.

3.3 Migration vers Rust 2021 : Descente aux Enfers de Cargo

L'exécution en mémoire de notre exécutable suit un article de fasterthanlime (https://fasterthanli.me/series/making-our-own-executable-packer) datant de quatre ans, rédigé pour l'édition Rust 2018 avec la toolchain "nightly-2021-02-14".
Notre code étant développé sous l'édition 2021, des problèmes de compatibilité se sont manifestés. Nous avons privilégié le portage du code de l'article vers l'édition 2021, plutôt que d'adapter notre projet à l'édition 2018. Malheureusement, cette migration s'est avérée complexe et ardue, soulignant les défis de l'écosystème Rust et de Cargo.

L'article original repose sur des modules qui ne sont plus maintenues, notament à `linked_list_allocator` qui dépend de `lock_api` qui n'est plus maintenue. Cela a engendré des problèmes de dépendances implicites obsolètes, souvent compliquées à identifier à cause des messages d'erreur de Cargo. On se retrouve contraints d'isoler et de tâtonner afin d'identifier le module qui utilise la dépendance. En effet, le compilateur Rust, bien que verbeux, produit souvent des messages d'erreur peu informatifs

Ainsi, la tentative de porter le projet vers l'édition 2021 a rapidement révélé, par le biais de Cargo, des problèmes de compatibilité et des régressions inattendues liés à cette transition d'édition.

La modernisation du projet s'est transformée en une véritable **chasse aux dépendances implicites**. De nombreuses bibliothèques ou fonctionnalités du langage, qui étaient disponibles ou se comportaient d'une certaine manière en 2018, ont subi des changements subtils mais significatifs dans l'édition 2021. Lorsque l'on réglait un problème 15 autres apparaissaient et actuellement, le projet se débat encore avec des **erreurs de linkage complexes**.

Bien que ces difficultés persistent, notre optimisme reste intact. Nous sommes confiants dans notre capacité à mener à bien ce projet. Si la migration du packer vers l'édition 2021 s'avère trop complexe, nous envisageront une réécriture complète, forte de l'expérience acquise grace aux articles.

En conclusion, cette tentative de portage a permis de révéler une facette moins reluisante de l'écosystème Rust. Si Cargo facilite grandement la gestion des dépendances dans de nombreux cas, il peut aussi se montrer impuissant face à des **changements profonds et non rétrocompatibles entre les éditions du langage**. On peut également mentionner crates.io, qui ne fournit pas l'information sur la version de Rust requise par chaque version d'une librairie. Ce qui nous contraints à procéder par tâtonnements, engendrant une perte de temps significative.

3.4 White Box

Pour la white box, nous avons voulu réimplémenter la white box **BVwhite box** développée par Lucas Barthélémy & Quarkslab : https://github.com/quarkslab/BVwhite box

Cette white box a été écrite en Python et utilise le module `numpy`.

De plus, le code, étant une white box, présente une complexité intrinsèque due à sa nature même et compte plus de 1000 lignes.

La porter en Rust a donc été compliqué. Cependant, nous avons réussi à implémenter les fonctions nécessaires au chiffrement et déchiffrement des fichiers.
Perdant trop de temps à essayer de porter `create_WB.py` en Rust, nous avons décidé de créer un processus Python qui génère la white box.
Cette décision est justifiée par le fait que ce script n'est utilisé qu'à la création du fichier packé et n'est pas requis lors de l'exécution du fichier packé lui-même.

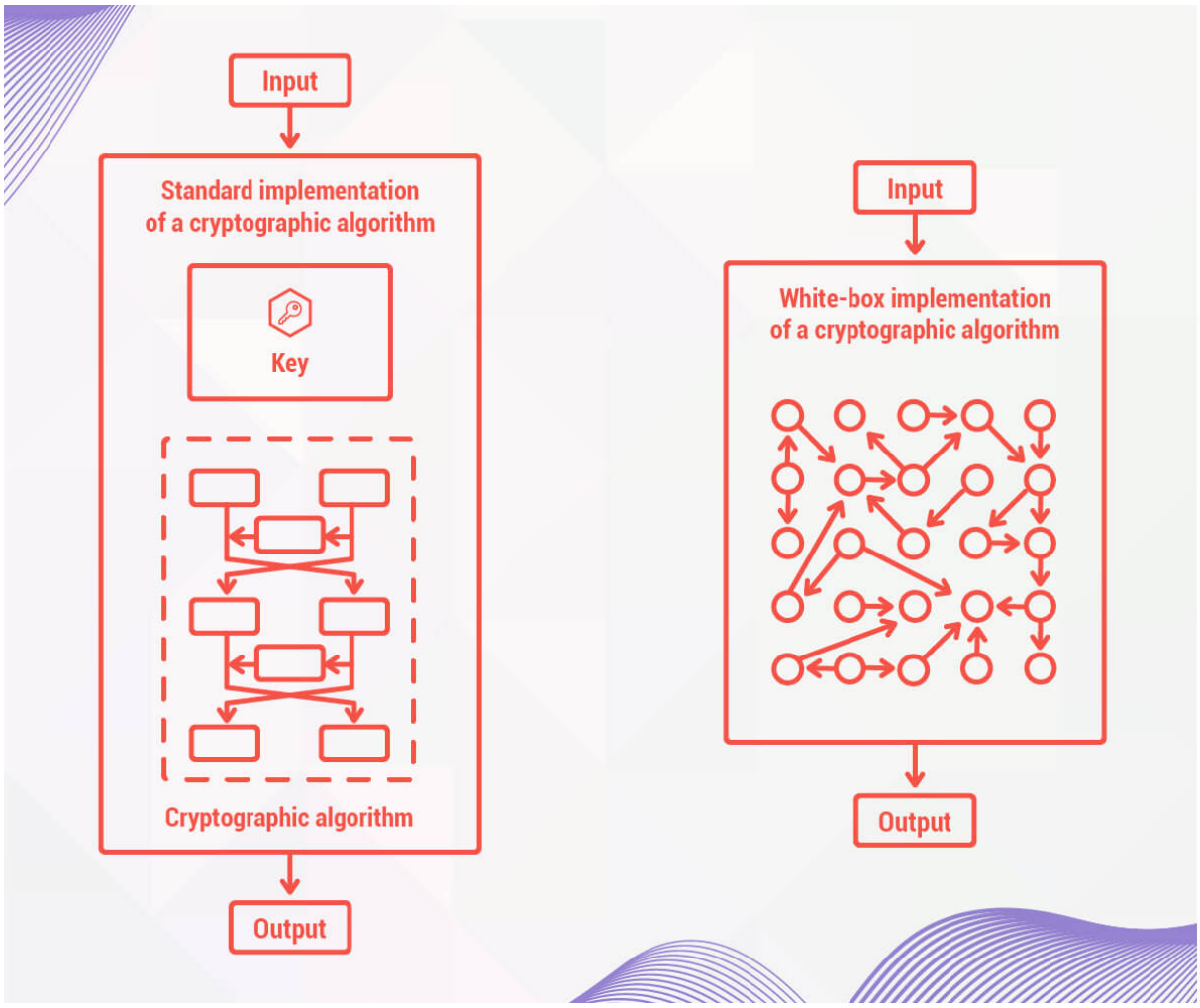


Figure 8: Implémentation whitebox expliquée

3.5 AES

L'implémentations d'AES 128 n'a pose quasiment aucun problème, mais nous avons du temporairement hardcoder la cle en raison des problèmes de dépendances mentionnes dans la partie 3.3.

Ces problèmes ne sont pas du fait de cette partie, mais ceci nous empêche d'importer le module random comme on le voudrait.

3.6 LZ77

Dans le cadre de l'optimisation des performances de compression des données, nous avons identifié la nécessité de remplacer l'algorithme de compression LZ77 par LZ4.

Le LZ77, bien qu'étant un algorithme de compression sans perte éprouvé, présente certaines limitations en termes de vitesse de compression et de décompression, ce qui peut impacter les performances globales du système, notamment pour les applications nécessitant un traitement en temps réel.

En effet, suite à notre changement de notre architecture de notre projet et du fonctionnement de notre logiciel, la vitesse de compression devient plus pertinente.
Avec la méthode de compression LZ77, nous avons pour but d'appliquer une compression simple et primordiale afin de pouvoir le coupler avec le codage de Huffman pour donner ce qui s'appelle "Deflate" en tant que bonus si le temps nous le permet.

Or, l'exécution se fera dans le fichier packé lui-même, et il est alors important de pouvoir exécuter un programme packé rapidement par la compression et la décompression de celui-ci.

Nous avons que de bonnes raisons de vouloir implémenter l'algorithme LZ4, accès aux données compressés rapide et complexité faible.

4 Répartition des tâches

Tâches	Louis	Arthur	Marcus	Gabriel
Compression		Suppléant		Délégué
Chiffrement			Délégué	Suppléant
White Box	Délégué		Suppléant	
Site Web		Délégué		Suppléant

Répartition des tâches effectuées entre les membres de l'équipe

5 Conclusion

Jusqu'ici, les tâches ont été réalisées dans les délais, bien que nous ayons rencontré bien plus de problèmes que prévu.

Au terme de cette première soutenance, nous avons pu mettre en place les bases essentielles de notre packer, avec une intégration réussie des fonctionnalités de compression et de chiffrement AES. Cependant, notre projet a rencontré plusieurs défis techniques, notamment liés à l'exécution en mémoire et aux limitations du langage Rust pour la manipulation bas niveau des binaires.

Dans les prochaines soutenances, notre objectif sera d'ajouter des mécanismes avancés, notamment des fonctionnalités d'anti-débogage, afin de complexifier davantage l'analyse et d'améliorer la robustesse de notre packer.

Malgré la difficulté de ce projet, nous sommes toujours motivés à le terminer, et à arriver à un produit fini de qualité !