ESISAR

– PX222 Maths/Info –

Implémentations d'AES en Haskell et en C

Jamyson Vetter - Siméon Studer
Mars - Juin 2026

Table des matières

1 Introduction et objectifs du projet **2**
1.1 Démarche de développement et jalons méthodologiques . . . 2

2 Fondements mathématiques et choix de modélisation **4**
2.1 Structures d'anneaux et de corps : classes de types `Anneau` et `Corps` . . . 4
2.2 Arithmétique modulaire sur les corps premiers $\mathbb{Z}/2\mathbb{Z}$ et $\mathbb{Z}/3\mathbb{Z}$ . . . 4
2.3 Anneau des polynômes génériques $K[X]$ et division euclidienne . . . 5
2.3.1 Normalisation, degré et égalité structurelle . . . 5
2.3.2 Opérations d'anneau sur $A[X]$ . . . 5
2.3.3 Division euclidienne dans $K[X]$ . . . 5
2.4 Construction du corps de Galois $\mathrm{GF}(2^8)$ . . . 6
2.5 Algorithme d'Euclide étendu, identité de Bézout et inversion dans $\mathrm{GF}(2^8)$ . . . 6
2.5.1 Inversion modulaire et normalisation unitaire (`inv_p`) . . . 7

3 Architecture logicielle et organisation des modules **8**
3.1 Présentation détaillée des couches et des fichiers . . . 8
3.2 Analyse du graphe de dépendances et prévention des cycles . . . 9

4 Implémentation de l'AES en Haskell **11**
4.1 Anneau quotient $\mathrm{GF}(2^8)[X]/(X^4+1)$ et représentation de l'état `State` . . . 11
4.2 Transformations de chiffrement et transformations inverses . . . 11
4.2.1 Substitution non linéaire : `subBytes` et `invSubBytes` . . . 12
4.2.2 Permutation des lignes : `shiftRows` et `invShiftRows` . . . 12
4.2.3 Diffusion intra-colonne : `mixColumns` et `invMixColumns` . . . 12
4.2.4 Addition de la sous-clé de ronde : `addRoundKey` et `invAddRoundKey` . . . 13
4.3 Planification de clés (`keyExpansion`) et prise en charge multi-tailles . . . 13
4.4 Orchestration du chiffrement (`cipher`), du déchiffrement (`invCipher`) et export C . . . 13

5 Stratégie de validation et conformité aux vecteurs NIST **15**
5.1 Architecture de la suite de tests GHCi . . . 15
5.2 Validation des propriétés algébriques et de la généricité ($\mathbb{Z}/2\mathbb{Z}$ et $\mathbb{Z}/3\mathbb{Z}$) . . . 15
5.3 Vérification des vecteurs de référence NIST FIPS 197 . . . 16

6 Prolongement expérimental : implémentation AES en C et modes opératoires **17**
6.1 Architecture du volet C et différences de représentation avec Haskell . . . 17
6.2 Déchiffrement direct (`InvCipher`) et déchiffrement équivalent (`EqInvCipher`) . . . 18
6.3 Modes opératoires ECB et CBC et gestion du bourrage PKCS#7 . . . 18
6.4 Analyse de l'entropie de Shannon sur l'image `Tux.bmp` . . . 19

7 Analyse des performances et optimisations **21**
7.1 Impact du choix de l'algorithme inverse et du recalcul de `KeyExpansion` . . . 21
7.2 Optimisation algorithmique de `MixColumns` et `InvMixColumns` . . . 26
7.3 Étude des optimisations du compilateur GCC (`-O0` à `-O3`) . . . 27

8 Conclusion et bilan technique **29**

1 Introduction et objectifs du projet

La cryptographie symétrique moderne repose sur l'articulation étroite entre des structures algébriques abstraites — principalement la théorie des anneaux et des corps finis [4] — et des contraintes d'ingénierie logicielle et matérielle fortes, telles que l'efficacité d'exécution, la résistance à la cryptanalyse et la conformité stricte aux standards internationaux. Adopté en 2001 par le *National Institute of Standards and Technology* (NIST) sous la référence FIPS PUB 197 [1], l'algorithme de chiffrement par blocs AES (*Advanced Encryption Standard*), issu de la primitive Rijndael conçue par Joan Daemen et Vincent Rijmen [2], constitue une illustration emblématique de cette dualité. Toutes ses transformations internes s'expriment en effet de manière algébrique dans le corps de Galois $\mathrm{GF}(2^8)$ et dans l'anneau quotient de polynômes $\mathrm{GF}(2^8)[X]/(X^4+1)$.

Ce rapport présente les travaux réalisés en binôme par Jamyson Vetter et Siméon Studer dans le cadre de l'unité d'enseignement PX222 Maths-Info à Grenoble INP – Esisar. L'objectif du projet consiste à concevoir, implémenter, valider et analyser une chaîne complète de chiffrement et de déchiffrement conforme à la norme AES (dans ses variantes à clés de 128, 192 et 256 bits) selon deux approches complémentaires et clairement différenciées :

1. **Un volet principal en langage fonctionnel Haskell** [3], dont la finalité dépasse la simple traduction impérative du pseudo-code de la norme FIPS 197. La démarche adoptée consiste à exploiter le système de types polymorphes et les classes de types (*typeclasses*) de Haskell afin de bâtir progressivement les abstractions algébriques générales — anneaux, corps commutatifs, anneau de polynômes $K[X]$, division euclidienne et algorithme d'Euclide étendu — avant de les spécialiser pour construire le corps $\mathrm{GF}(2^8)$ et les transformations de l'AES.
2. **Un volet expérimental et système en langage C**, dédié à l'étude pratique du chiffrement de données réelles, à l'implémentation des modes opératoires de chiffrement par blocs ECB (*Electronic Codebook*) et CBC (*Cipher Block Chaining*) avec bourrage PKCS#7, à l'évaluation statistique de l'entropie de Shannon sur des fichiers d'images matricielles (`Tux.bmp`), ainsi qu'au profilage fin des performances et à l'optimisation algorithmique et compilatoire.

1.1 Démarche de développement et jalons méthodologiques

Le développement du projet s'est inscrit dans une méthodologie incrémentale et collaborative, appuyée sur une gestion de versions sous Git et rythmée par deux grandes phases : quatre séances de conception et d'implémentation fonctionnelle en Haskell (du 13 mars au 6 mai 2026), suivies d'une semaine de développement intensif en langage C (du 8 au 12 juin 2026). L'analyse rétrospective du carnet de bord met en évidence une progression structurée autour de cinq jalons techniques majeurs :

- **Jalon 1 – Socle algébrique initial (Séance 1, 13/03/2026) :** définition collégiale de la classe de types `Anneau`, choix de la représentation des polynômes par listes de coefficients ordonnés selon les puissances croissantes, et instanciation initiale de la structure d'anneau pour le type polymorphe `Polynome a`.
- **Jalon 2 – Arithmétique polynomiale et corps fini** $\mathrm{GF}(2^8)$ **(Séance 2, 01/04/2026) :** répartition modulaire des développements algorithmiques. Jamyson Vetter a conçu et implémenté la multiplication polynomiale, la division euclidienne (`de_pol`) ainsi que l'affichage personnalisé des polynômes via la classe `Show`. Parallèlement, Siméon Studer a formalisé et implémenté l'algorithme d'Euclide étendu (`euclide`), l'inversion modulaire (`inv_p`), ainsi que le type `GF28` muni de sa réduction modulo le polynôme irréductible $m(x)$ de l'AES. Cette phase a également permis au binôme de maîtriser la résolution de conflits de fusion (*merge*) sous Git lors de la mise en commun des briques algébriques.
- **Jalon 3 – Restructuration modulaire et transformations AES (Séance 3, 21/04/2026) :** afin d'éliminer les risques de couplage fort et de dépendances circulaires, Jamyson Vetter a procédé en amont de la séance à une refonte architecturale séparant le code en sept modules distincts (`Anneau.hs`, `Z2.hs`, `Polynome.hs`, `GF28.hs`, `Euclide.hs`, `AES.hs`, `Main.hs`), complétés par `SBox.hs`. En séance, Jamyson Vetter a développé les transformations d'état directes (`subBytes`,

`shiftRows`, `mixColumns`) et l'ensemble des transformations inverses (`invSubBytes`, `invShiftRows`, `invMixColumns`, `invAddRoundKey`), tout en corrigeant une boucle infinie dans `mixColumns` par l'adaptation de la division euclidienne aux polynômes à coefficients dans GF(2^8) et la redéfinition structurelle de l'égalité `Eq`. Siméon Studer a implémenté `addRoundKey`, la génération du plan de clés (`keyExpansion`), la modélisation de l'anneau quotient `GF28X` et de la matrice d'état `State`, ainsi que les conversions de blocs.

— **Jalon 4 – Généralisation mathématique, orchestration et conformité NIST (Séance 4, 06/05/2026 et finalisation) :** Siméon Studer a épuré l'implémentation de `AES.hs`, corrigé la taille des polynômes issus de la multiplication dans `GF28X` (`mul_gf28x` via `take 4`, conservant exactement quatre coefficients), développé les outils de formatage hexadécimal (`gf28ToHex`, `afficher_key`, `afficher_state`), achevé l'orchestration du déchiffrement (`invCipher`) et validé le passage à l'échelle sur les clés de 192 et 256 bits.De son côté, Jamyson Vetter a corrigé un défaut de transposition matricielle dans `state_vers_octets` (passage d'une lecture ligne par ligne à une reconstruction colonne par colonne), généralisé toute l'arithmétique polynomiale à un corps quelconque par l'ajout de la soustraction `sub` dans `Anneau`, la normalisation unitaire du PGCD dans `inv_p` et l'introduction du corps $\mathbb{Z}/3\mathbb{Z}$ (`Z3.hs`), puis structuré l'ensemble des bancs de tests automatisés (`tests_*.ghci`) et la documentation architecturale.

— **Jalon 5 – Implémentation C, modes opératoires et profilage (08/06/2026 – 12/06/2026) :** développement en langage C du moteur AES complet (incluant `InvCipher` et `EqInvCipher`), des modes ECB et CBC appliqués au chiffrement d'images BMP, mesure de l'entropie de Shannon, analyse des goulots d'étranglement avec *Valgrind* et *KCachegrind*, optimisation de `MixColumns` et `InvMixColumns`, et étude de l'impact des niveaux d'optimisation du compilateur GCC (`-O0` à `-O3`) par inspection du code assembleur via `objdump`.

2 Fondements mathématiques et choix de modélisation

L'architecture développée en Haskell traduit directement la hiérarchie des structures algébriques nécessaires à la définition de l'AES [2, 4]. Plutôt que de manipuler des octets sous forme d'entiers machine figés, chaque entité cryptographique est modélisée par son statut algébrique exact.

2.1 Structures d'anneaux et de corps : classes de types `Anneau` et `Corps`

En algèbre générale, un anneau commutatif unitaire $(A, +, \times)$ est un ensemble muni d'une loi d'addition interne formant un groupe abélien (d'élément neutre 0_A et où tout élément admet un opposé, ce qui définit la soustraction) et d'une loi de multiplication associative, commutative, distributive par rapport à l'addition et possédant un élément neutre 1_A. Lorsque $1_A \neq 0_A$ et que tout élément non nul de A admet en outre un inverse multiplicatif dans A, la structure $(A, +, \times)$ constitue un corps commutatif.

Dans le module fondamental `Anneau.hs`, ces deux concepts sont modélisés à l'aide du mécanisme des classes de types de Haskell [3] :

- La classe `Anneau a` déclare les constantes `zero :: a` (0_A) et `un :: a` (1_A), ainsi que les opérations binaires `add :: a -> a -> a`, `sub :: a -> a -> a` et `mul :: a -> a -> a`.
- La classe `Corps a` hérite par contrainte de contexte de `Anneau a` (`class (Anneau a) => Corps a`) et y adjoint l'opération d'inversion multiplicative `inv :: a -> a`.

Évolution architecturale – Généralisation de la soustraction : *Dans une première version centrée exclusivement sur la caractéristique 2 (où l'addition et la soustraction coïncident, puisque $+1 \equiv -1 \pmod 2$), l'opération `sub` n'était pas présente dans la classe `Anneau`. Lors de la généralisation des algorithmes polynomiaux à des corps finis quelconques en séance 4, l'introduction explicite de `sub` dans `Anneau` s'est révélée indispensable : tant l'étape de réduction de la division euclidienne que la mise à jour des coefficients de Bézout dans l'algorithme d'Euclide étendu reposent sur une soustraction algébrique stricte.*

2.2 Arithmétique modulaire sur les corps premiers $\mathbb{Z}/2\mathbb{Z}$ et $\mathbb{Z}/3\mathbb{Z}$

Pour instancier les structures polynomiales, deux corps finis premiers de cardinal p ($\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$) ont été implémentés :

1. **Le corps binaire $\mathbb{Z}/2\mathbb{Z}$ (`Z2.hs`) :** le type algébrique `data Z2 = Z2 Integer deriving (Eq)` encapsule un entier relatif. Pour deux représentants canoniques dans $\{0, 1\}$, les opérations de l'instance `Anneau Z2` vérifient :

$$\forall (a, b) \in (\mathbb{Z}/2\mathbb{Z})^2, \quad a + b = a - b = (a + b) \bmod 2, \quad a \times b = (a \cdot b) \bmod 2.$$

   L'addition (et la soustraction, définie par `sub = add`) correspond au « ou exclusif » logique (XOR), tandis que la multiplication correspond au « et » logique (AND, calculé directement par `a * b` puisque le produit sur $\{0, 1\}$ demeure réduit). L'instance `Corps Z2` définit `inv (Z2 1) = Z2 1` et lève une erreur explicite sur l'élément nul `Z2 0`. Le module fournit également une instance `Anneau Integer` ainsi qu'une instance personnalisée `Show Z2` (via `show_z2`) qui affiche directement la valeur entière sous-jacente (`0` ou `1`) sans le constructeur syntaxique `Z2`, allégeant ainsi considérablement la lecture des polynômes.

2. **Le corps ternaire $\mathbb{Z}/3\mathbb{Z}$ (`Z3.hs`) :** construit selon le même principe (`data Z3 = Z3 Integer deriving (Eq)`), ce module implémente les opérations `add`, `sub` et `mul` avec une réduction modulo 3. En caractéristique 3, la soustraction se distingue de l'addition ($1 - 2 \equiv 2 \pmod 3$ tandis que $1 + 2 \equiv 0 \pmod 3$) et le groupe multiplicatif $(\mathbb{Z}/3\mathbb{Z})^* = \{1, 2\}$ possède un élément non trivial dont l'inverse est lui-même (`inv (Z3 2) = Z3 2`, car $2 \times 2 = 4 \equiv 1 \pmod 3$). Ce module constitue le socle de validation de la généricité des algorithmes polynomiaux.

2.3 Anneau des polynômes génériques $K[X]$ et division euclidienne

Le module `Polynome.hs` modélise l'anneau des polynômes $A[X]$ à coefficients dans un anneau quelconque A via le type paramétrique :

```
data Polynome a = Polynome [a]
```

Une liste $[a_0, a_1, a_2, \ldots, a_n]$ représente le polynôme ordonné selon les puissances croissantes de l'indéterminée X :

$$P(X) = \sum_{k=0}^{n} a_k X^k = a_0 + a_1 X + a_2 X^2 + \cdots + a_n X^n.$$

2.3.1 Normalisation, degré et égalité structurelle

La représentation d'un polynôme par une liste finie de coefficients introduit une ambiguïté canonique : l'ajout de zéros en fin de liste (coefficients nuls pour les degrés supérieurs, tels que $[a_0, a_1, 0_A, 0_A]$) ne modifie pas l'objet mathématique mais altère la structure de la liste Haskell. Lors des opérations de soustraction ou de division euclidienne, l'annulation des termes de plus haut degré génère naturellement de tels zéros résiduels.

Pour résoudre ce problème, la fonction `nettoyer :: (Anneau a, Eq a) => [a] -> [a]` supprime les éléments nuls en fin de liste en inversant la liste, en éliminant les éléments égaux à `zero` en tête, puis en la réinversant :

```
nettoyer l = reverse $ dropWhile (== zero) $ reverse l
```

Cette normalisation intervient dans deux mécanismes essentiels :

- **Égalité algébrique (`instance (Anneau a, Eq a) => Eq (Polynome a)`) :** deux polynômes sont déclarés égaux si et seulement si leurs listes de coefficients nettoyées sont identiques (`nettoyer l1 == nettoyer l2`).
- **Calcul du degré (`degre :: (Anneau a, Eq a) => Polynome a -> Int`) :** le degré est évalué sur la liste normalisée $l' =$ `nettoyer l`. Par convention d'implémentation, le polynôme nul (l' vide) reçoit le degré -1, tandis qu'un polynôme non nul a pour degré `length` $l' - 1$.

2.3.2 Opérations d'anneau sur $A[X]$

Pour tout type `a` instance de `Anneau`, le type `Polynome a` est lui-même instancié dans la classe `Anneau` :

- Les éléments neutres sont définis par `zero = Polynome [zero]` et `un = Polynome [un]`.
- L'addition (`addp`, appuyée sur `addL`) et la soustraction (`subp`, appuyée sur `subL`) procèdent terme à terme sur les listes de coefficients. Lorsque le premier polynôme est plus court que le second dans `subL [] ys`, les coefficients restants de `ys` sont remplacés par leurs opposés respectifs via `map (sub zero) ys`.
- La multiplication (`mulp`, appuyée sur `mulL`) réalise le produit de convolution discret par récurrence sur la liste des coefficients du premier polynôme :

  $$(x + X \cdot P_{\text{suite}}(X)) \cdot Q(X) = x \cdot Q(X) + X \cdot (P_{\text{suite}}(X) \cdot Q(X)),$$

  ce qui se traduit directement en Haskell par `addL (map (\y -> mul x y) ys) (zero : mulL xs ys)`.

2.3.3 Division euclidienne dans $K[X]$

Lorsque l'ensemble des coefficients est un corps K (contrainte de type `(Corps a, Eq a)`), l'anneau $K[X]$ est un anneau euclidien [4]. Pour tout couple de polynômes $(A, B) \in K[X]^2$ avec $B \neq 0$, il existe un unique couple $(Q, R) \in K[X]^2$ (quotient et reste) vérifiant :

$$A(X) = B(X)Q(X) + R(X) \quad \text{avec} \quad \deg(R) < \deg(B).$$

La fonction `de_pol :: (Corps a, Eq a) => Polynome a -> Polynome a -> (Polynome a, Polynome a)` implémente l'algorithme de division posée par récurrence sur le degré du dividende :

1. Si $\deg(B) = -1$, l'évaluation est interrompue par une erreur (`"Division par un polynôme nul impossible"`).
2. Si $\deg(A) = -1$, le couple (`zero`, `zero`) est retourné ; si $\deg(A) < \deg(B)$, la division est terminée et renvoie (`zero`, A).
3. Sinon, soient $d_A = \deg(A)$, $d_B = \deg(B)$ et $\Delta = d_A - d_B \geq 0$. La fonction extrait les coefficients dominants $c_A = A_{d_A}$ (`listA !! dA`) et $c_B = B_{d_B}$ (`listB !! dB`), puis calcule le coefficient dominant du quotient :

$$c_Q = c_A \cdot c_B^{-1} \quad (\texttt{coeffQuotient = mul coeffA (inv coeffB)}).$$

4. Elle construit alors le monôme $Q_{\text{actuel}}(X) = c_Q X^{\Delta}$ (`replicate diff zero ++ [coeffQuotient]`), forme le polynôme décalé $B_{\text{décalé}}(X) = B(X) \cdot Q_{\text{actuel}}(X)$, et élimine le terme de plus haut degré de A en calculant $A_{\text{réduit}}(X) = A(X) - B_{\text{décalé}}(X)$ (`sub a bDecale`).
5. Un appel récursif `de_pol aReduit b` fournit le quotient complémentaire Q_{suite} et le reste final R_{final}, permettant de retourner $(Q_{\text{actuel}} + Q_{\text{suite}}, R_{\text{final}})$.

2.4 Construction du corps de Galois $\mathrm{GF}(2^8)$

Dans le standard AES [1], chaque octet $b = \{b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0\}$ est identifié à un élément du corps fini à $2^8 = 256$ éléments, noté $\mathrm{GF}(2^8)$ ou $\mathbb{F}_{2^8}$. Algébriquement, ce corps d'extension est construit comme l'anneau quotient de $\mathbb{F}_2[x]$ par l'idéal maximal engendré par un polynôme irréductible $m(x)$ de degré 8 sur $\mathbb{F}_2$ [4] :

$$\mathrm{GF}(2^8) \simeq \frac{(\mathbb{Z}/2\mathbb{Z})[x]}{\big(m(x)\big)}, \quad \text{où} \quad m(x) = x^8 + x^4 + x^3 + x + 1.$$

Les éléments de $\mathrm{GF}(2^8)$ sont ainsi représentés de manière unique par les polynômes à coefficients dans $\mathbb{Z}/2\mathbb{Z}$ de degré strictement inférieur à 8 :

$$b(x) = \sum_{i=0}^{7} b_i x^i, \quad b_i \in \mathbb{Z}/2\mathbb{Z}.$$

Dans le module `GF28.hs`, l'alias `type GF2X = Polynome Z2` et le type `data GF28 = GF28 GF2X deriving (Show, Eq)` concrétisent cette représentation. Le polynôme irréductible $m(x)$ (correspondant à la notation hexadécimale $\{01\}\{1b\}$) est défini selon l'ordre des puissances croissantes par :

```
m = Polynome [Z2 1, Z2 1, Z2 0, Z2 1, Z2 1, Z2 0, Z2 0, Z2 0, Z2 1].
```

Le type `GF28` est instancié dans la classe `Anneau` :

— L'addition `add_gf28` (ainsi que la soustraction `sub = add_gf28`) additionne directement les polynômes sous-jacents dans $\mathbb{F}_2[x]$, sans réduction nécessaire puisque le degré de la somme demeure inférieur ou égal à 7.

— La multiplication `mul_gf28` calcule le produit polynomial $a(x) \cdot b(x)$ dans $\mathbb{F}_2[x]$, puis en extrait le reste de la division euclidienne par $m(x)$ à l'aide de `de_pol (mul a b) m`.

2.5 Algorithme d'Euclide étendu, identité de Bézout et inversion dans $\mathrm{GF}(2^8)$

Pour ériger l'anneau commutatif `GF28` en un corps (`instance Corps GF28`), il est nécessaire de calculer l'inverse multiplicatif de tout élément non nul $P(x) \in \mathrm{GF}(2^8)$ modulo le polynôme irréductible $m(x)$. Le module `Euclide.hs` résout ce problème de manière générique pour tout corps de base K grâce à l'algorithme d'Euclide étendu.

Soient $A, B \in K[X]$ deux polynômes. L'algorithme d'Euclide étendu détermine le plus grand commun diviseur $D = \mathrm{PGCD}(A, B)$ ainsi qu'un couple de polynômes de Bézout $(U, V) \in K[X]^2$ vérifiant l'identité de Bézout :

$$A(X)U(X) + B(X)V(X) = \mathrm{PGCD}(A, B).$$

La fonction `euclide :: (Corps a, Eq a) => (Polynome a, Polynome a) -> (Polynome a, Polynome a, Polynome a)` s'appuie sur la fonction récursive terminale `aux` qui maintient l'invariant :

$$\forall k \geq -1, \quad A \cdot u_k + B \cdot v_k = r_k,$$

avec l'initialisation $(r_{-1}, r_0, u_{-1}, u_0, v_{-1}, v_0) = (A, B, 1, 0, 0, 1)$. Tant que le reste courant r_k est non nul, la division euclidienne `de_pol` r_{k-1} r_k fournit le quotient q_k et le nouveau reste r_{k+1}, et les suites de Bézout sont mises à jour selon les relations de récurrence :

$$\begin{aligned} r_{k+1} &= r_{k-1} - q_k \cdot r_k, \\ u_{k+1} &= u_{k-1} - q_k \cdot u_k \quad (\texttt{u_nv = sub u' (mul q u)}), \\ v_{k+1} &= v_{k-1} - q_k \cdot v_k \quad (\texttt{v_nv = sub v' (mul q v)}). \end{aligned}$$

Lorsque $r_k = 0$, l'algorithme s'arrête et retourne le triplet $(r_{k-1}, u_{k-1}, v_{k-1})$.

2.5.1 Inversion modulaire et normalisation unitaire (`inv_p`)

Lorsque $m(X)$ est irréductible et que $P(X) \not\equiv 0 \pmod{m(X)}$ avec $\deg(P) < \deg(m)$, les polynômes P et m sont premiers entre eux. Le dernier reste non nul $D = r_{k-1}$ retourné par `euclide (p, m)` est alors un polynôme constant non nul $D(X) = c \in K^*$. L'identité de Bézout s'écrit :

$$P(X)U(X) + m(X)V(X) = c \implies P(X) \cdot \left(c^{-1}U(X)\right) \equiv 1 \pmod{m(X)}.$$

Dans le cas particulier de $\mathbb{Z}/2\mathbb{Z}$, la seule constante non nulle est $c = 1$, de sorte que $U(X)$ est directement l'inverse cherché. En revanche, sur un corps quelconque tel que $\mathbb{Z}/3\mathbb{Z}$ ou $\mathrm{GF}(2^8)$, l'algorithme d'Euclide produit en général un PGCD constant non unitaire ($c \neq 1$). La fonction `inv_p` procède donc à une normalisation explicite : elle nettoie la liste des coefficients de `pgcd`, extrait le scalaire constant c (`last l_net`) et multiplie le polynôme de Bézout $U(X)$ par le polynôme constant $[c^{-1}]$ (`u_norm = mul u (Polynome [inv c])`). L'instanciation finale du corps de Galois s'écrit alors directement dans `Euclide.hs` :

```
instance Corps GF28 where inv (GF28 p) = GF28 (inv_p p m).
```

3 Architecture logicielle et organisation des modules

L'architecture logicielle du volet Haskell a été conçue pour garantir une séparation stricte entre les mécanismes algébriques génériques et la spécification cryptographique de l'AES. Le code source est structuré en quatre couches hiérarchiques dont chaque composant assume une responsabilité unique.

3.1 Présentation détaillée des couches et des fichiers

Les tableaux 1, 2 et 3 détaillent le rôle et le contenu de l'ensemble des fichiers du dépôt Haskell ainsi que des outils de suivi.

Fichier	Rôle	Description
`Anneau.hs`	**Interfaces algébriques de base**	Définit les classes de types fondamentales `Anneau` (éléments neutres `zero`, `un`, et opérations `add`, `sub`, `mul`) et `Corps` (héritant de `Anneau` avec l'opération `inv`).
`Z2.hs`	**Arithmétique modulo 2**	Fournit le type `Z2` et définit ses instances `Anneau`, `Corps` et `Show`, ainsi qu'une instance `Anneau` pour `Integer`. `Z2` sert de base aux coefficients des polynômes utilisés dans `GF28`.
`Z3.hs`	**Tests de généricité**	Fournit le type `Z3` et ses instances `Anneau`, `Corps` et `Show` pour l'arithmétique modulo 3, permettant de vérifier que les algorithmes polynomiaux ne dépendent pas uniquement du cas binaire.

Table 1 – Modules de la Couche 1 : Fondations algébriques.

Fichier	Rôle	Description
`Polynome.hs`	**Calcul polynomial générique**	Définit la structure `Polynome a`, les opérations d'anneau (`addp`, `subp`, `mulp`), le degré (`degre`), la division euclidienne (`de_pol`) et redéfinit `Eq` via `nettoyer`.
`GF28.hs`	**Structure du corps fini**	Déclare l'alias `GF2X` (`Polynome Z2`), le type `GF28` modélisant $\mathrm{GF}(2^8)$, le polynôme irréductible `m` et l'instance `Anneau GF28` (addition et multiplication modulo m).
`Euclide.hs`	**Algorithmes de résolution**	Implémente l'algorithme d'Euclide étendu (`euclide`), le calcul d'inverse modulo m normalisé (`inv_p`) sur tout type `Corps`, et déclare l'instance finale `Corps GF28`.

Table 2 – Modules de la Couche 2 (Arithmétique des polynômes) et de la Couche 3 (Corps de Galois).

Fichier	Rôle	Description
`SBox.hs`	**Données statiques**	Contient la table de substitution (`sbox`) de 256 octets de l'AES, conservée comme implémentation alternative statique face au calcul algébrique dynamique.
`AES.hs`	**Moteur AES complet**	Définit l'anneau `GF28X`, l'état `State`, l'expansion de clé (`keyExpansion`), les transformations directes et inverses, ainsi que `cipher` et `invCipher` pour AES-128, 192 et 256.
`Main.hs`	**Point d'entrée IO**	Importe les différents modules de l'architecture afin d'exécuter le projet au sein de la monade `IO`.
`CDB.md`	**Carnet de bord**	Documente le suivi chronologique des séances, la répartition des tâches, les décisions architecturales et les campagnes expérimentales en C.

Table 3 – Modules de la Couche 4 (Application AES et exécution) et documentation de suivi.

3.2 Analyse du graphe de dépendances et prévention des cycles

La figure 1 représente le graphe complet des dépendances entre les modules Haskell du projet, structuré selon les quatre couches d'abstraction. Par convention, une flèche orientée de A vers B ($A \to B$) indique que le module B dépend des définitions exportées par le module A.

Cette organisation répond à trois exigences majeures d'ingénierie logicielle :

1. **Polymorphisme et indépendance de la Couche 2 :** le module `Polynome.hs` dépend exclusivement des abstractions déclarées dans `Anneau.hs`. Il n'importe ni `Z2.hs` ni `Z3.hs`. Grâce au polymorphisme paramétrique contraint par les classes de types, le code de la division euclidienne (`de_pol`) s'applique sans aucune modification aux polynômes sur $\mathbb{Z}/2\mathbb{Z}$ (pour construire $\mathrm{GF}(2^8)$), aux polynômes sur $\mathbb{Z}/3\mathbb{Z}$ (pour les tests de généricité) et aux polynômes à coefficients dans $\mathrm{GF}(2^8)$ (pour construire l'anneau $\mathrm{GF}(2^8)[X]/(X^4+1)$ utilisé dans `mixColumns`).
2. **Découplage de la Couche 3 et élimination des dépendances circulaires :** en Haskell, l'importation mutuelle entre deux modules provoque un cycle de compilation. Si `GF28.hs` devait définir immédiatement l'instance `Corps GF28`, il devrait importer `Euclide.hs` pour disposer de `inv_p` ; or `Euclide.hs` a besoin de `GF28.hs` pour connaître le polynôme irréductible `m` et le type `GF28`. Le découpage adopté résout élégamment cette contrainte : `GF28.hs` définit le type `GF28`, le polynôme `m` et l'instance `Anneau GF28`, tandis que `Euclide.hs` importe `GF28.hs`, implémente `euclide` et `inv_p`, et déclare l'instance finale `Corps GF28`.
3. **Isolation de la Couche 4 :** le module `AES.hs` consomme le corps fini fourni par la Couche 3 sans exposer les détails de réduction polynomiale de bas niveau. Le module `SBox.hs` demeure quant à lui totalement autonome, séparant la table de constantes statiques de la logique de calcul algébrique.

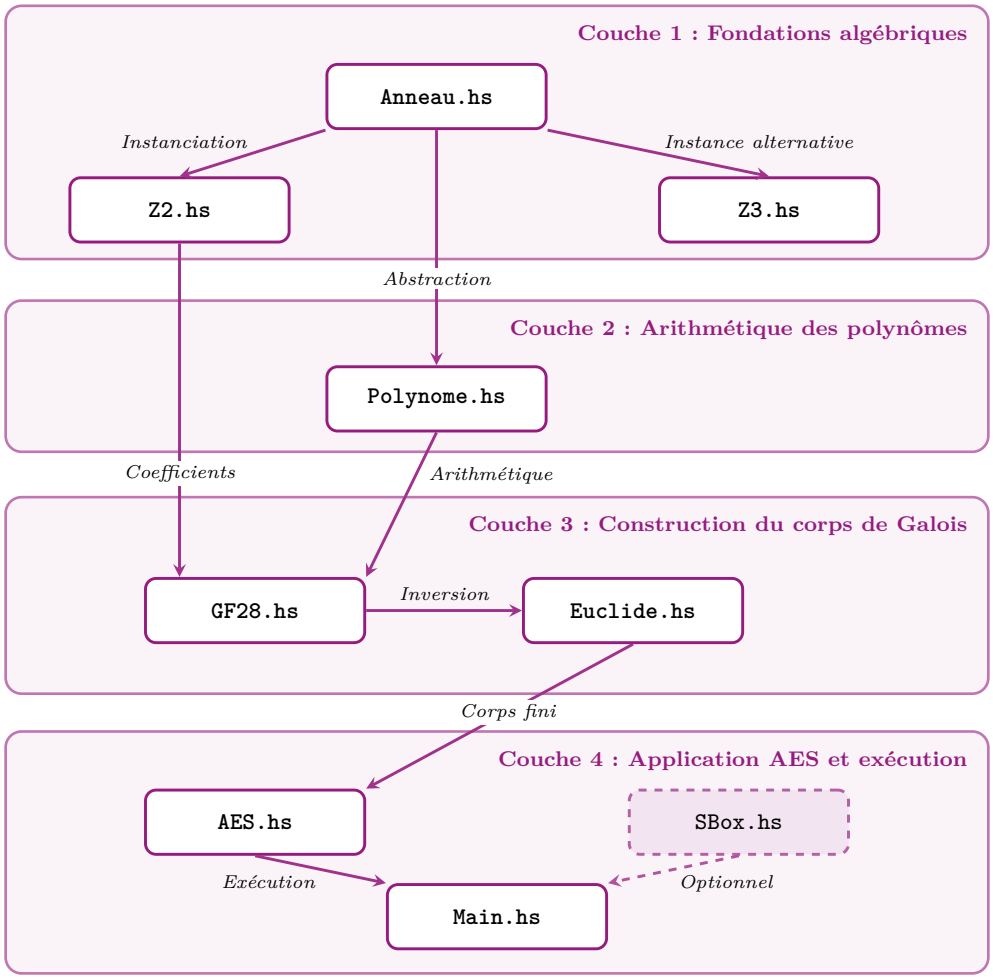


Figure 1 – Architecture logicielle en couches et graphe de dépendances des modules Haskell. Le sens des flèches ($A \to B$) indique que le module B dépend du module A.

4 Implémentation de l'AES en Haskell

Le module `AES.hs` réalise l'intégralité des spécifications du standard FIPS 197 [1] en s'appuyant sur les structures algébriques des couches inférieures.

4.1 Anneau quotient $\mathrm{GF}(2^8)[X]/(X^4+1)$ et représentation de l'état `State`

Dans la norme AES, l'état interne (*State*) est une matrice de $4 \times Nb$ octets (avec $Nb = 4$, soit 16 octets ou 128 bits). Or, les opérations de chiffrement telles que `MixColumns` et `AddRoundKey` traitent chacune des 4 colonnes de l'état comme un mot de 32 bits assimilé à un polynôme de degré au plus 3 à coefficients dans $\mathrm{GF}(2^8)$:

$$s(X) = s_3X^3 + s_2X^2 + s_1X + s_0 \in \mathrm{GF}(2^8)[X].$$

Pour modéliser cette structure de second niveau, le module `AES.hs` introduit le type `GF28X` représentant l'anneau quotient $\mathrm{GF}(2^8)[X]/(X^4+1)$:

```
data GF28X = GF28X (Polynome GF28) deriving (Show, Eq).
```

Le polynôme de réduction de degré 4, $m_{\mathrm{GF28X}}(X) = X^4 + 1$, est défini par :

```
m_GF28X = Polynome [un, zero, zero, zero, un].
```

`GF28X` est instancié dans la classe `Anneau` avec l'addition polynomiale `add_gf28x` (qui réalise un XOR octet par octet entre deux colonnes) et la multiplication modulaire `mul_gf28x` :

```
mul_gf28x (GF28X a) (GF28X b) = GF28X (Polynome (take 4 r))
    where (_, Polynome r) = de_pol (mul a b) m_GF28X
```

Point de vigilance technique – Maintien d'une taille fixe et anneau non intègre : *Contrairement à $m(x)$ sur $\mathbb{F}_2$, le polynôme $X^4+1 = (X+1)^4$ n'est pas irréductible sur* $\mathrm{GF}(2^8)$ *: la structure* ***`GF28X`*** *est donc un anneau commutatif mais n'est pas un corps. Néanmoins, son coefficient dominant étant égal à* ***`un`*** *($1_{\mathrm{GF}(2^8)}$), qui est inversible dans le corps* ***`GF28`****, la fonction générique* ***`de_pol`*** *s'applique parfaitement pour réduire le produit modulo X^4+1. En outre, comme la soustraction polynomiale dans* ***`de_pol`*** *conserve en fin de liste les coefficients nuls des degrés 4 à 6 annulés lors de la réduction, l'application de* ***`take 4 r`*** *dans* ***`mul_gf28x`*** *garantit que le polynôme retourné possède toujours exactement 4 éléments, prévenant ainsi tout échec de filtrage par motif (pattern matching) dans les transformations ultérieures.*

La matrice d'état est alors définie par les synonymes de types :

```
type Colonne = GF28X,     type State = [Colonne].
```

Les conversions entre un bloc linéaire de 16 octets (`[GF28]`) et la matrice d'état (`State`) respectent la convention d'indexation par colonnes de la norme FIPS 197 ($s[r,c] = \mathrm{in}[r+4c]$ pour $0 \le r, c < 4$) :

- `octets_vers_state :: [GF28] -> [GF28X]` regroupe les 16 octets d'entrée par paquets consécutifs de 4 pour former les 4 colonnes $[c_0, c_1, c_2, c_3]$.
- `state_vers_octets :: [GF28X] -> [GF28]` déballe les 4 colonnes $c_j = [s_{0,j}, s_{1,j}, s_{2,j}, s_{3,j}]$ et restitue la liste linéaire $[s_{0,0}, s_{1,0}, s_{2,0}, s_{3,0}, s_{0,1}, \ldots, s_{3,3}]$. Les conversions élémentaires entre un entier (`Int`) et un octet algébrique (`GF28`) sont assurées par `intToGf28` (via `toZ2List`) et `gf28ToInt`.

4.2 Transformations de chiffrement et transformations inverses

Chaque ronde de l'AES s'appuie sur quatre transformations inversibles agissant sur l'état `State` [1, 2].

4.2.1 Substitution non linéaire : `subBytes` et `invSubBytes`

La transformation `subBytes` est l'unique brique non linéaire de l'AES. Elle applique indépendamment à chaque octet de l'état (via `subBytes = map subBytesCol` et `subByte`) la boîte de substitution (S-Box), définie comme la composée de deux opérations :

1. **Inversion multiplicative dans** $\mathrm{GF}(2^8)$ **(`inv_gf28`) :** tout octet non nul est remplacé par son inverse multiplicatif calculé dynamiquement par l'algorithme d'Euclide étendu (`inv octet`), tandis que l'élément nul $\{00\}$ est envoyé sur lui-même (`zero`).
2. **Transformation affine sur** $\mathbb{F}_2$ **(`sub_affine`) :** à l'octet inversé de bits $(b_0, \ldots, b_7)$ est appliquée la relation :

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i, \quad 0 \le i < 8,$$

où $c = \{63\}_{16} = \{01100011\}_2$. Plutôt que de recourir à des manipulations de bits impératives, `sub_affine` tire parti du caractère circulant de la matrice affine : l'opération équivaut exactement à une multiplication polynomiale dans l'anneau $\mathbb{F}_2[X]/(X^8+1)$ par le polynôme :

$$a_{\text{aff}}(X) = 1 + X + X^2 + X^3 + X^4 \quad (\texttt{a = Polynome [un, un, un, un, un]})$$

modulo $m_8(X) = X^8 + 1$, suivie de l'addition du polynôme constant $c(X) = 1 + X + X^5 + X^6$ (`Polynome [un, un, zero, zero, zero, un, un, zero]`) et d'un formatage sur 8 bits (`take 8 (coeffs_rc ++ repeat zero)`).

Pour le déchiffrement, `invSubBytes` (via `invSubBytesCol` et `invSubByte`) applique les opérations réciproques dans l'ordre inverse : d'abord la transformation affine inverse `invSub_affine`, réalisée par multiplication modulo $X^8 + 1$ par :

$$a_{\text{inv}}(X) = X + X^3 + X^6 \quad (\texttt{Polynome [zero, un, zero, un, zero, zero, un, zero]})$$

suivie de l'addition de $c_{\text{inv}}(X) = 1 + X^2$ ($\{05\}_{16}$), puis l'inversion multiplicative `inv_gf28`.

Parallèlement à ce calcul algébrique intégral, le module `SBox.hs` (ainsi qu'un bloc commenté dans `AES.hs` utilisant `mapGF28X`) conserve une implémentation statique par table de correspondance (`sbox :: [Int]`).

4.2.2 Permutation des lignes : `shiftRows` et `invShiftRows`

La fonction `shiftRows :: State -> State` assure la diffusion inter-colonnes en décalant circulairement vers la gauche les octets de la ligne r ($0 \le r < 4$) de r positions :

$$s'_{r,c} = s_{r,(c+r) \bmod 4}.$$

L'état étant structuré par colonnes $[c_0, c_1, c_2, c_3]$, `shiftRows` extrait par filtrage de motif les 16 octets $s_{r,c}$ et recompose les quatre nouvelles colonnes : par exemple, la première colonne devient $[s_{0,0}, s_{1,1}, s_{2,2}, s_{3,3}]$. La transformation inverse `invShiftRows :: State -> State` réalise le décalage circulaire vers la droite de r positions (soit $4 - r$ positions vers la gauche), reconstruisant la première colonne sous la forme $[s_{0,0}, s_{1,3}, s_{2,2}, s_{3,1}]$.

4.2.3 Diffusion intra-colonne : `mixColumns` et `invMixColumns`

La transformation `mixColumns :: State -> State` opère sur chaque colonne vue comme un élément de l'anneau $\mathrm{GF}(2^8)[X]/(X^4+1)$ en la multipliant modulo $X^4 + 1$ par le polynôme fixe inversible :

$$a(X) = \{03\}X^3 + \{01\}X^2 + \{01\}X + \{02\} \quad (\texttt{a_mix}).$$

Grâce à l'instanciation de `GF28X` dans la classe `Anneau`, cette opération matricielle complexe s'écrit en une seule ligne fonctionnelle :

```
mixColumns = map (\col -> a_mix `mul` col).
```

De manière symétrique, la transformation inverse `invMixColumns :: State -> State` multiplie chaque colonne dans `GF28X` par le polynôme inverse $a^{-1}(X)$ vérifiant $a(X) \otimes a^{-1}(X) \equiv \{01\} \pmod{X^4+1}$:

$$a^{-1}(X) = \{0b\}X^3 + \{0d\}X^2 + \{09\}X + \{0e\} \quad (\texttt{a_inv_mix}).$$

4.2.4 Addition de la sous-clé de ronde : `addRoundKey` et `invAddRoundKey`

La fonction `addRoundKey :: State -> State -> State` combine récursivement chaque colonne de l'état avec le mot correspondant de la sous-clé de ronde à l'aide de l'addition `add_gf28x` de l'anneau `GF28X`. L'addition dans $\mathbb{Z}/2\mathbb{Z}$ étant involutive ($x \oplus k \oplus k = x$), la transformation inverse est strictement identique : `invAddRoundKey = addRoundKey`.

4.3 Planification de clés (`keyExpansion`) et prise en charge multi-tailles

L'algorithme AES se décline en trois variantes selon la longueur de la clé secrète initiale :

- **AES-128 :** clé de 16 octets ($Nk = 4$ mots de 32 bits), $Nr = 10$ rondes, soit $Nb(Nr+1) = 44$ mots générés.
- **AES-192 :** clé de 24 octets ($Nk = 6$ mots), $Nr = 12$ rondes, soit 52 mots générés.
- **AES-256 :** clé de 32 octets ($Nk = 8$ mots), $Nr = 14$ rondes, soit 60 mots générés.

La fonction `keyExpansion :: [GF28] -> Int -> Int -> [GF28X]` prend en entrée la clé sous forme d'octets ainsi que les paramètres Nk et Nr, et génère la liste complète des mots de sous-clés $w[0 \dots 4(Nr+1)-1]$. Elle s'appuie sur :

- `rotWord :: GF28X -> GF28X`, qui effectue une permutation circulaire $[a_0, a_1, a_2, a_3] \mapsto [a_1, a_2, a_3, a_0]$;
- `subWord :: Colonne -> Colonne`, synonyme de `subBytesCol` ;
- `rconList :: [GF28X]`, liste des constantes de ronde $\mathrm{Rcon}[i] = [x^{i-1} \bmod m(x), \{00\}, \{00\}, \{00\}]$ pour $1 \le i \le 10$ (avec $\{01\}, \{02\}, \{04\}, \{08\}, \{10\}, \{20\}, \{40\}, \{80\}, \{1b\}, \{36\}$) ;
- trois fonctions auxiliaires reproduisant la structure itérative de la norme : `af` (affectation fonctionnelle dans une liste), `f1` (copie des Nk premiers mots de la clé initiale) et le couple mutuellement récursif `g1`/`f2` qui calcule pour $i \ge Nk$ le mot $w[i] = w[i-Nk] \oplus \mathrm{temp}'$, où, en notant $t = w[i-1]$:

$$\mathrm{temp}' = \begin{cases} \texttt{subWord}(\texttt{rotWord}(t)) \oplus \mathrm{Rcon}[i/Nk] & \text{si } i \bmod Nk = 0, \\ \texttt{subWord}(t) & \text{si } Nk > 6 \text{ et } i \bmod Nk = 4, \\ t & \text{sinon,} \end{cases}$$

la seconde branche correspondant à la substitution intermédiaire spécifique à la variante AES-256.

4.4 Orchestration du chiffrement (`cipher`), du déchiffrement (`invCipher`) et export C

La fonction `intervale_4 :: Int -> [a] -> [a]` extrait la sous-clé de 4 mots associée à la ronde i (`take 4 (drop (i * 4) w)`). L'orchestration complète suit rigoureusement la spécification FIPS 197 :

- **Chiffrement (`cipher :: Int -> [GF28] -> [GF28X] -> [GF28]`) :** convertit les 16 octets d'entrée en `State` (`octets_vers_state`), applique `addRoundKey` avec la sous-clé de ronde 0, exécute les $Nr-1$ rondes complètes via la fonction récursive `for_round` (enchaînant `subBytes`, `shiftRows`, `mixColumns` et `addRoundKey`), applique la ronde finale Nr dépourvue de `mixColumns` (`addRoundKey ((shiftRows . subBytes) state2) (intervale_4 nr w)`), puis reconvertit l'état final en liste d'octets (`state_vers_octets`).
- **Déchiffrement (`invCipher :: [GF28] -> [GF28X] -> Int -> Int -> [GF28]`) :** s'appuie sur `invCipher_state`, qui additionne la dernière sous-clé (ronde Nr), exécute à rebours de $Nr-1$ jusqu'à 1 la boucle `for_round_inv` dont chaque pas (`fff`) applique successivement `invShiftRows`, `invSubBytes`, `addRoundKey` et `invMixColumns`, puis termine hors boucle par `hors_boucle_inv_cipher` (`invShiftRows`, `invSubBytes` et `addRoundKey` avec la sous-clé 0).

— **Affichage et interopérabilité vers C :** outre les fonctions d'inspection hexadécimale (`gf28ToHex`, `afficher_key`, `afficher_state`), le module fournit les fonctions `gF28_vers_C`, `export_key`, `extraire_ligne`, `pexport_state` et `export_state`. Ces dernières génèrent automatiquement des chaînes de caractères formatées selon la syntaxe des tableaux 1D et 2D du langage C (par exemple `"{{0x39, 0x02, ...}}"`), ce qui a permis d'exporter directement les états de référence calculés en Haskell vers les bancs de tests unitaires du volet C (`test_aes.c`).

5 Stratégie de validation et conformité aux vecteurs NIST

Afin de garantir l'exactitude mathématique et la conformité cryptographique de l'implémentation Haskell, une stratégie de validation ascendante (*bottom-up*) a été mise en place. Chaque couche architecturale fait l'objet de tests unitaires dédiés avant l'exécution des tests d'intégration sur les vecteurs officiels du standard NIST FIPS 197 [1].

5.1 Architecture de la suite de tests GHCi

Le tableau 4 présente les cinq scripts de validation interprétés sous GHCi qui composent le banc d'essai du projet.

Fichier	Rôle	Description
`tests_anneaux.ghci`	**Validation algébrique de base**	Teste via `runTestsAnneaux` l'arithmétique sur `Z2`, les opérations sur les listes (`addL`, `mulL`) et les polynômes (`addp`, `mulp`, `degre`, `de_pol`), `euclide` et la multiplication dans `GF28`.
`tests_corps_quelconque.ghci`	**Généricité sur $\mathbb{F}_2$ et $\mathbb{F}_3$**	Teste via `runTestsQuelconque` la division euclidienne et l'algorithme d'Euclide étendu sur `Z3` et `Z2` (incluant $\deg(A) < \deg(B)$) et vérifie systématiquement l'identité de Bézout.
`tests_fonctions_AES.ghci`	**Tests unitaires AES**	Contrôle via `runTestsAES` les conversions hexadécimales, les bijections matricielles, `rotWord`, `subWord`, la longueur de `keyExpansion` (44, 52, 60 mots) et chaque transformation directe et inverse.
`tests_chiffrement.ghci`	**Conformité NIST FIPS 197**	Valide via `runTestsChiffrement` les annexes A, B et C de la norme (AES-128, 192, 256), les états intermédiaires de ronde, les tests *Round-Trip* et les fonctions d'export vers C.
`tests_complets.ghci`	**Régression globale**	Point d'entrée unique exécutant séquentiellement les quatre scripts précédents pour certifier l'intégrité globale du dépôt après toute modification.

Table 4 – Organisation des scripts de validation GHCi du volet Haskell.

5.2 Validation des propriétés algébriques et de la généricité ($\mathbb{Z}/2\mathbb{Z}$ et $\mathbb{Z}/3\mathbb{Z}$)

La validation des couches basses procède en deux étapes :

1. **Vérification des opérations dans $\mathbb{F}_2[X]$ et $\mathrm{GF}(2^8)$ (`tests_anneaux.ghci`) :** le script contrôle les lois de `Z2` ($1+1=0$, $1\times 1=1$), la convolution polynomiale (notamment le produit de $1+X^3$ par X^4+X^7 donnant X^4+X^{10}), la division euclidienne de ce produit par le polynôme irréductible $m(X)$ (quotient $Q(X)=X^2$ et reste $R(X)=X^2+X^3+X^4+X^5+X^6$), ainsi que la concordance exacte de ce reste avec la multiplication directe `mul gf28_a gf28_b` dans `GF28`.
2. **Démonstration de la généricité sur $\mathbb{Z}/3\mathbb{Z}$ (`tests_corps_quelconque.ghci`) :** quatre scénarios mettent à l'épreuve le polymorphisme de `Polynome.hs` et `Euclide.hs` :
   — *Cas 1 (Division basique dans $\mathbb{F}_3[X]$) :* pour $A(X)=2X^2+X+1$ et $B(X)=X+2$, `de_pol a b` retourne le quotient attendu $Q(X)=2X$ (`[0, 2]`) et le reste $R(X)=1$ (`[1]`).
   — *Cas 2 (Degré supérieur dans $\mathbb{F}_3[X]$) :* division de $C(X)=X^3+2X^2+X+1$ par $D(X)=2X^2+1$, validant le quotient $Q(X)=2X+1$ (`[1, 2]`) et le reste $R(X)=2X$ (`[0, 2]`).

— *Cas 3 (Dividende de degré strictement inférieur au diviseur) :* la division de $B(X)$ (degré 1) par $A(X)$ (degré 2) retourne immédiatement $Q(X) = 0$ et $R(X) = B(X)$.
— *Cas 4 (Changement de corps vers $\mathbb{F}_2[X]$) :* division de $E(X) = X^3 + X + 1$ par $F(X) = X + 1$, fournissant $Q(X) = X^2 + X$ (`[0, 1, 1]`) et $R(X) = 1$.

Dans chacun de ces quatre cas, le triplet $(\text{pgcd}, U, V) =$ `euclide`(P_1, P_2) est soumis au test algébrique formel de l'identité de Bézout :

```
add (mul p1 u) (mul p2 v) == pgcd,
```

confirmant que l'implémentation de la soustraction `sub` et de la division générique respecte les invariants euclidiens indépendamment de la caractéristique du corps.

5.3 Vérification des vecteurs de référence NIST FIPS 197

Les scripts `tests_fonctions_AES.ghci` et `tests_chiffrement.ghci` confrontent l'implémentation aux vecteurs normatifs des annexes A, B et C de la publication FIPS 197 [1] :

— **Annexe A — Expansion de clés (*Key Expansion*) :** la fonction `keyExpansion` est évaluée sur les trois clés de référence `key_A1` (128 bits, commençant par `2b 7e 15 16...`), `key_A2` (192 bits, `8e 73 b0 f7...`) et `key_A3` (256 bits, `60 3d eb 10...`). Outre la vérification du nombre exact de mots générés (respectivement 44, 52 et 60 mots), les transformations élémentaires sont validées pas à pas sur le mot $w_3 =$ `09 cf 4f 3c` de l'annexe A.1 : `rotWord` produit `cf 4f 3c 09` et `subWord` produit `8a 84 eb 01`, en parfait accord avec la norme.
— **Annexe B — États intermédiaires du chiffrement AES-128 :** à partir du bloc d'entrée `input_B` (`32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34`) et de la clé étendue issue de `key_A1`, le banc de test vérifie non seulement le chiffré final à l'issue des 10 rondes (`39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32`), mais contrôle également l'état exact de la matrice 4×4 (donné ci-dessous en lecture ligne par ligne) à chaque étape interne de la première ronde :

1. Après `addRoundKey` initial (`state_B_init`) : `19 a0 9a e9 | 3d f4 c6 f8 | e3 e2 8d 48 | be 2b 2a 08` ;
2. Après `subBytes` (`state_B_sub`) : `d4 e0 b8 1e | 27 bf b4 41 | 11 98 5d 52 | ae f1 e5 30` ;
3. Après `shiftRows` (`state_B_shift`) : `d4 e0 b8 1e | bf b4 41 27 | 5d 52 11 98 | 30 ae f1 e5` ;
4. Après `mixColumns` (`state_B_mix`) : `04 e0 48 28 | 66 cb f8 06 | 81 19 d3 26 | e5 9a 7a 4c`.

De plus, l'application de chaque transformation inverse (`invMixColumns`, `invShiftRows`, `invSubBytes`, `invAddRoundKey`) sur l'état correspondant de l'annexe B permet de vérifier unitairement que chaque opération annule exactement l'effet de sa contrepartie directe.
— **Annexe C — Chiffrement, déchiffrement multi-tailles et tests *Round-Trip* :** sur le texte clair de référence `00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff`, le script valide les trois variantes complètes de la norme :
— *Annexe C.1 (AES-128, $Nk = 4, Nr = 10$)* avec la clé `00 01 ... 0f` → chiffré obtenu : `69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a` ;
— *Annexe C.2 (AES-192, $Nk = 6, Nr = 12$)* avec la clé `00 01 ... 17` → chiffré obtenu : `dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91` ;
— *Annexe C.3 (AES-256, $Nk = 8, Nr = 14$)* avec la clé `00 01 ... 1f` → chiffré obtenu : `8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89`.

Enfin, les tests d'intégration de type *Round-Trip* vérifient pour l'ensemble des vecteurs des annexes B, C.1, C.2 et C.3 la relation fondamentale d'inversion :

```
invCipher (cipher nr input w) w 4 nr == input.
```

6 Prolongement expérimental : implémentation AES en C et modes opératoires

En complément de la modélisation fonctionnelle en Haskell, une seconde implémentation complète a été réalisée en langage C dans le répertoire `AES_en_c/`. Ce volet vise à étudier le comportement de l'AES lorsqu'il est confronté au chiffrement de fichiers volumineux, à mettre en œuvre les modes opératoires ECB et CBC, et à analyser les performances d'exécution à bas niveau.

6.1 Architecture du volet C et différences de représentation avec Haskell

Le tableau 5 synthétise l'organisation des fichiers sources qui composent le répertoire `AES_en_c/`.

Fichier	Rôle	Description
`AES.h / AES.c`	**Cœur cryptographique AES**	Implémente les transformations AES, `xtime()`, `KeyExpansion`, `EqKeyExpansion`, le chiffrement `Cipher` et les deux variantes de déchiffrement `InvCipher` et `EqInvCipher`.
`ECB.h / ECB.c`	**Mode Electronic Codebook**	Implémente `aes_ecb_encrypt` et `aes_ecb_decrypt` avec bourrage PKCS#7 et vérification d'intégrité au déchiffrement.
`CBC.h / CBC.c`	**Mode Cipher Block Chaining**	Implémente `aes_cbc_encrypt` et `aes_cbc_decrypt` avec vecteur d'initialisation (`iv`) de 16 octets, chaînage XOR et bourrage PKCS#7.
`bmp.h / bmp.c`	**Manipulation d'images BMP**	Isole l'en-tête (`header`) et le tableau de pixels (`pixels`) d'un fichier BMP via la structure `chunk` pour chiffrer uniquement les données visuelles.
`chiffrement_tux_ecb.c`	**Banc d'essai visuel ECB**	Chiffre et déchiffre les pixels de `Tux.bmp` en mode ECB en mesurant le temps d'exécution et le débit en octets par seconde.
`chiffrement_tux_cbc.c`	**Banc d'essai visuel CBC**	Chiffre et déchiffre les pixels de `Tux.bmp` en mode CBC avec mesure de temps et de débit.
`entropie.c`	**Analyse statistique**	Calcule l'entropie de Shannon (en bits par octet) de la distribution des octets de pixels d'une image BMP.
`test_aes.c / test_ecb.c / test_cbc.c`	**Validation unitaire C**	Vérifie la conformité NIST de `AES.c` (128, 192, 256 bits) et teste les modes ECB et CBC sur des tailles multiples (32 octets) et non multiples de 16 (20 et 43 octets).

Table 5 – Organisation des fichiers sources de l'implémentation en langage C (`AES_en_c/`).

Sur le plan architectural, l'implémentation C adopte des choix de représentation adaptés à l'exécution impérative en mémoire contiguë :

— Les types de base sont définis dans `AES.h` avec la constante `NB = 4` par `typedef unsigned char byte`, `typedef byte word[NB]` et `typedef byte state[NB][NB]` (où `s[r][c]` désigne l'octet à la ligne `r` et à la colonne `c`). L'utilisation du type non signé `unsigned char` garantit un comportement déterministe des opérateurs de décalage binaire (`<<` et `>>`).

— Les substitutions `SubBytes` et `InvSubBytes` s'appuient sur les tables statiques précalculées `sbox[256]` et `inv_sbox[256]` par accès direct en complexité $\mathcal{O}(1)$.
— La multiplication dans GF(2^8) repose sur l'opération élémentaire `xtime(byte x)` décrite à la section 4.2.1 de la norme FIPS 197 [1] : la multiplication d'un octet par le monôme x ($\{02\}_{16}$) se traduit par un décalage binaire d'un bit vers la gauche, suivi d'un XOR conditionnel avec $\{1b\}_{16}$ (les 8 bits de poids faible de $m(x)$) si le bit de poids fort b_7 valait 1 avant décalage :

```
return (x << 1) ^ (((x & 0x80) != 0) ? 0x1b : 0x00);
```

6.2 Déchiffrement direct (`InvCipher`) et déchiffrement équivalent (`EqInvCipher`)

Outre l'algorithme de déchiffrement direct `InvCipher` (qui inverse l'ordre des transformations en enchaînant `InvShiftRows`, `InvSubBytes`, `AddRoundKey` puis `InvMixColumns`), le module `AES.c` implémente la variante `EqInvCipher` (*Equivalent Inverse Cipher*, section 5.3.5 de FIPS 197 [1]).

Cette variante repose sur deux propriétés algébriques de l'AES [2] :

1. **Commutativité de `InvSubBytes` et `InvShiftRows` :** la substitution agissant octet par octet indépendamment de leur position et le décalage ne déplaçant que les indices sans modifier la valeur des octets, on a $\texttt{InvShiftRows} \circ \texttt{InvSubBytes} = \texttt{InvSubBytes} \circ \texttt{InvShiftRows}$.
2. **Linéarité de `InvMixColumns` sur** GF(2^8) **:** la multiplication matricielle étant linéaire par rapport à l'addition (XOR), on dispose pour tout état S et toute sous-clé de ronde K_r de l'identité :

$$\texttt{InvMixColumns}(S \oplus K_r) = \texttt{InvMixColumns}(S) \oplus \texttt{InvMixColumns}(K_r).$$

En appliquant préalablement `InvMixColumns` aux sous-clés intermédiaires des rondes 1 à $Nr-1$ lors de l'expansion de clé — ce que réalise la fonction `EqKeyExpansion` en produisant le tableau de clés modifiées `dw[]` —, l'algorithme `EqInvCipher` conserve exactement la même séquence structurelle d'opérations que `Cipher` (`InvSubBytes`, `InvShiftRows`, `InvMixColumns`, puis `AddRoundKey` avec `dw`).

Résolution d'un problème d'indexation mémoire dans `EqKeyExpansion` : *En langage C, la matrice `state s[4][4]` est indexée par `s[ligne][colonne]`, tandis que le tableau de sous-clés `word dw[NB*(Nr+1)]` stocke chaque colonne de 4 octets de manière contiguë (`dw[colonne][ligne]`). Appliquer directement la fonction `InvMixColumns` par transtypage sur `&dw[round*NB]` revenait donc à mélanger les lignes de la sous-clé au lieu de ses colonnes. L'identification de cette inversion mémoire a conduit à écrire la fonction dédiée `InvMixColumns_eq(word s[NB])`, qui transpose les indices d'accès (`s[c][0..3]`) lors de la préparation de `dw[]`.*

6.3 Modes opératoires ECB et CBC et gestion du bourrage PKCS#7

La primitive AES opérant strictement sur des blocs de 16 octets, le chiffrement d'un flux de données de longueur arbitraire L requiert un schéma de bourrage (*padding*) et un mode opératoire :

— **Bourrage PKCS#7 :** dans `ECB.c` et `CBC.c`, la longueur de bourrage est calculée par $\ell_{\text{pad}} = 16 - (L \bmod 16) \in \{1, \ldots, 16\}$. Le tampon alloué de taille $L + \ell_{\text{pad}}$ (multiple exact de 16) reçoit les L octets clairs suivis de ℓ_{pad} octets valant chacun ℓ_{pad}. Au déchiffrement, la valeur du dernier octet indique ℓ_{pad} : les fonctions `aes_ecb_decrypt` et `aes_cbc_decrypt` vérifient systématiquement que $1 \le \ell_{\text{pad}} \le 16$ et que les ℓ_{pad} derniers octets portent tous cette valeur exacte, libérant la mémoire et retournant un code d'erreur (`-1`) en cas d'altération.
— **Mode ECB (*Electronic Codebook*, `ECB.c`) :** chaque bloc clair P_i de 16 octets est converti en matrice `state` colonne par colonne ($r = k \bmod 4$, $c = \lfloor k/4 \rfloor$ pour $0 \le k < 16$) et chiffré indépendamment des autres :

$$C_i = E_K(P_i), \qquad P_i = D_K(C_i).$$

— **Mode CBC (*Cipher Block Chaining*, `CBC.c`) :** pour briser le déterminisme inter-blocs, chaque bloc clair P_i est combiné par un XOR bit à bit avec le bloc chiffré précédent C_{i-1} avant d'entrer dans le chiffreur AES, en initialisant la chaîne avec un vecteur d'initialisation $C_0 = \text{IV}$ de 16 octets :

$$C_i = E_K(P_i \oplus C_{i-1}) \quad \text{avec } C_0 = \text{IV}, \qquad P_i = D_K(C_i) \oplus C_{i-1}.$$

6.4 Analyse de l'entropie de Shannon sur l'image `Tux.bmp`

Afin d'évaluer expérimentalement les propriétés de diffusion des modes ECB et CBC, les programmes `chiffrement_tux_ecb.c` et `chiffrement_tux_cbc.c` s'appuient sur le module `bmp.c` pour conserver l'en-tête du fichier `Tux.bmp` intact tout en chiffrant son tableau de pixels. Le programme `entropie.c` mesure ensuite l'entropie empirique d'ordre 1 de Shannon de la séquence des octets de pixels :

$$H(X) = -\sum_{i=0}^{255} P(x_i) \log_2 P(x_i), \quad \text{où } P(x_i) = \frac{N(x_i)}{N_{\text{total}}}.$$

Pour une source émettant des octets sur un alphabet de 256 symboles équiprobables ($P(x_i) = 1/256$), l'entropie théorique maximale vaut :

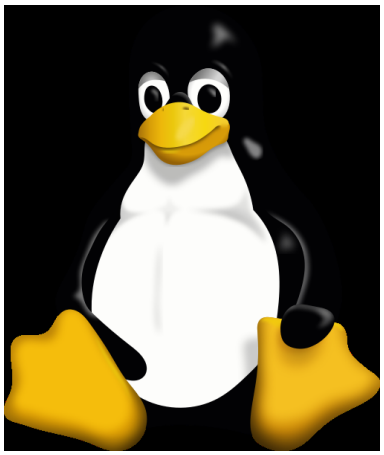
$$H_{\max} = \log_2(256) = 8 \text{ bits par octet.}$$

Le tableau 6 regroupe les mesures obtenues sur l'image originale et sur les images chiffrées.

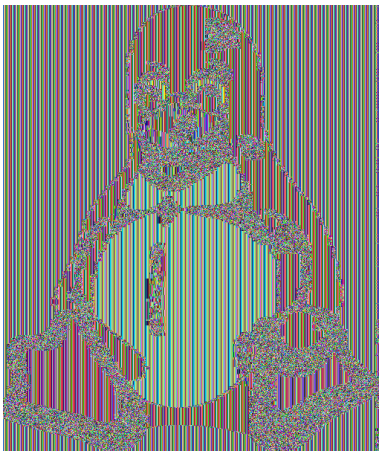
Configuration de l'image	Fichier analysé	Entropie (bits/octet)
Image originale non chiffrée	`Tux.bmp`	3,938782
Image chiffrée en mode ECB	`Tux_chiffre_256_ecb.bmp`	6,994900
Image chiffrée en mode CBC	`Tux_chiffre_cbc_128.bmp`	7,999785

Table 6 – Mesures de l'entropie de Shannon sur les pixels de l'image `Tux.bmp` avant et après chiffrement.

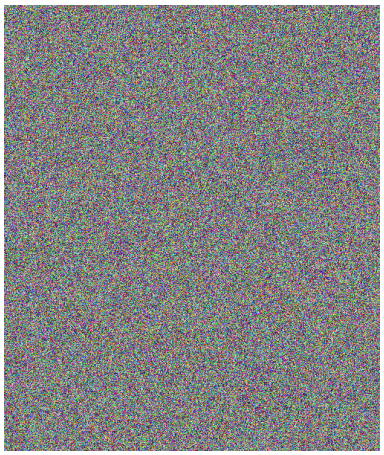
La figure 2 illustre visuellement la différence de diffusion entre les modes ECB et CBC sur les pixels de l'image `Tux.bmp`.



(a) Image originale
($H = 3{,}939$ bits/octet)



(b) Mode ECB
($H = 6{,}995$ bits/octet)



(c) Mode CBC
($H = 7{,}9998$ bits/octet)

Figure 2 – Comparaison visuelle du chiffrement de `Tux.bmp` : conservation de la silhouette en mode ECB (b) et disparition complète des motifs en mode CBC (c).

L'analyse de ces résultats met en évidence trois phénomènes distincts :

1. **Redondance spatiale de l'image source ($H \approx 3{,}939$ bits/octet) :** l'image `Tux.bmp` comporte de vastes aplats de couleur uniforme (fond blanc, zones noires et jaunes du manchot), ce qui entraîne une forte concentration de l'histogramme sur un nombre restreint de valeurs d'octets et explique la faible entropie initiale.
2. **Limitation structurelle du mode ECB ($H \approx 6{,}995$ bits/octet) :** bien que la bijection non linéaire d'AES augmente sensiblement l'entropie par rapport à l'image claire, le mode ECB est déterministe à l'échelle d'un bloc de 16 octets ($P_i = P_j \implies C_i = C_j$). Les zones uniformes de l'image originale engendrent des milliers de blocs de 16 octets identiques qui sont transformés en blocs chiffrés identiques : la silhouette du manchot demeure visuellement reconnaissable et la distribution statistique des octets reste éloignée de l'uniformité ($H \approx 7{,}0 < 8$).
3. **Diffusion inter-blocs du mode CBC ($H \approx 7{,}999785$ bits/octet) :** grâce à l'addition modulo 2 avec le bloc chiffré précédent avant chaque appel à `Cipher`, deux blocs clairs identiques $P_i = P_j$ reçoivent des masques pseudo-aléatoires distincts $C_{i-1} \neq C_{j-1}$ et produisent des blocs chiffrés décorrélés. Les motifs visuels disparaissent intégralement au profit d'un bruit visuellement uniforme, et l'entropie mesurée converge à $2{,}15 \times 10^{-4}$ bit/octet près vers le maximum théorique $H_{\max} = 8$.

Rigueur méthodologique — Portée et limites de l'entropie de Shannon : *Il convient de distinguer rigoureusement l'observation statistique expérimentale de la garantie théorique de sécurité cryptographique. L'obtention d'une entropie d'ordre 1 très proche de 8 bits par octet atteste uniquement de l'équirépartition empirique des fréquences d'apparition des 256 valeurs d'octets dans le fichier chiffré ; elle ne constitue nullement une preuve suffisante de sécurité sémantique (IND-CPA) ni d'indiscernabilité calculatoire avec une séquence véritablement aléatoire. À titre d'illustration, un fichier constitué de la simple répétition déterministe de la séquence ordonnée* $(0, 1, 2, \ldots, 255)$ *présenterait une entropie de Shannon exactement égale à 8 bits par octet, tout en ayant une sécurité cryptographique nulle. De même, la sécurité du mode CBC en conditions réelles ne dépend pas uniquement de la disparition des motifs visuels sur une image, mais exige impérativement que le vecteur d'initialisation (*IV*) soit généré de manière aléatoire et imprévisible à chaque nouveau chiffrement.*

7 Analyse des performances et optimisations

La seconde partie des expérimentations menées en langage C porte sur l'évaluation quantitative des performances temporelles lors du chiffrement et du déchiffrement de l'image `Tux.bmp` (environ 890,7 Ko), le profilage dynamique de l'exécution à l'aide des outils *Valgrind* (module *Callgrind*) et *KCachegrind*, l'optimisation algorithmique des transformations de colonnes, et l'étude de la génération de code par le compilateur GCC.

7.1 Impact du choix de l'algorithme inverse et du recalcul de `KeyExpansion`

Quatre configurations initiales ont été instrumentées et profilées afin d'isoler deux facteurs : d'une part, la comparaison entre le déchiffrement équivalent (`EqInvCipher`) et le déchiffrement direct (`InvCipher`) ; d'autre part, le coût d'un recalcul de l'expansion de clé à chaque bloc de 16 octets (appels répétés à `aes_encrypt`, `aes_decrypt` et `aes_decrypt2`) par rapport à un précalcul unique de la clé étendue en début de traitement. Le tableau 7 synthétise les temps d'exécution et les débits mesurés sur ces quatre configurations avant optimisation de `MixColumns`.

Stratégie d'expansion de clé	Variante	Opération	Temps (s)	Débit (o/s)
KeyExpansion unique	`EqInvCipher`	Chiffrement	0,143 s	6 367 982 o/s
KeyExpansion unique	`EqInvCipher`	Déchiffrement	0,745 s	1 224 169 o/s
KeyExpansion unique	`InvCipher`	Chiffrement	0,144 s	6 332 605 o/s
KeyExpansion unique	`InvCipher`	Déchiffrement	0,735 s	1 240 321 o/s
KeyExpansion répétée / bloc	`EqInvCipher`	Chiffrement	0,209 s	4 366 737 o/s
KeyExpansion répétée / bloc	`EqInvCipher`	Déchiffrement	1,440 s	633 278 o/s
KeyExpansion répétée / bloc	`InvCipher`	Chiffrement	0,209 s	4 354 955 o/s
KeyExpansion répétée / bloc	`InvCipher`	Déchiffrement	0,797 s	1 144 319 o/s

Table 7 – Comparaison des performances initiales selon la variante de déchiffrement (`EqInvCipher` ou `InvCipher`) et la stratégie de calcul de `KeyExpansion`.

Les graphes d'appels générés avec *Valgrind* et *KCachegrind* pour ces quatre configurations sont présentés sur les figures 3, 4, 5 et 6.

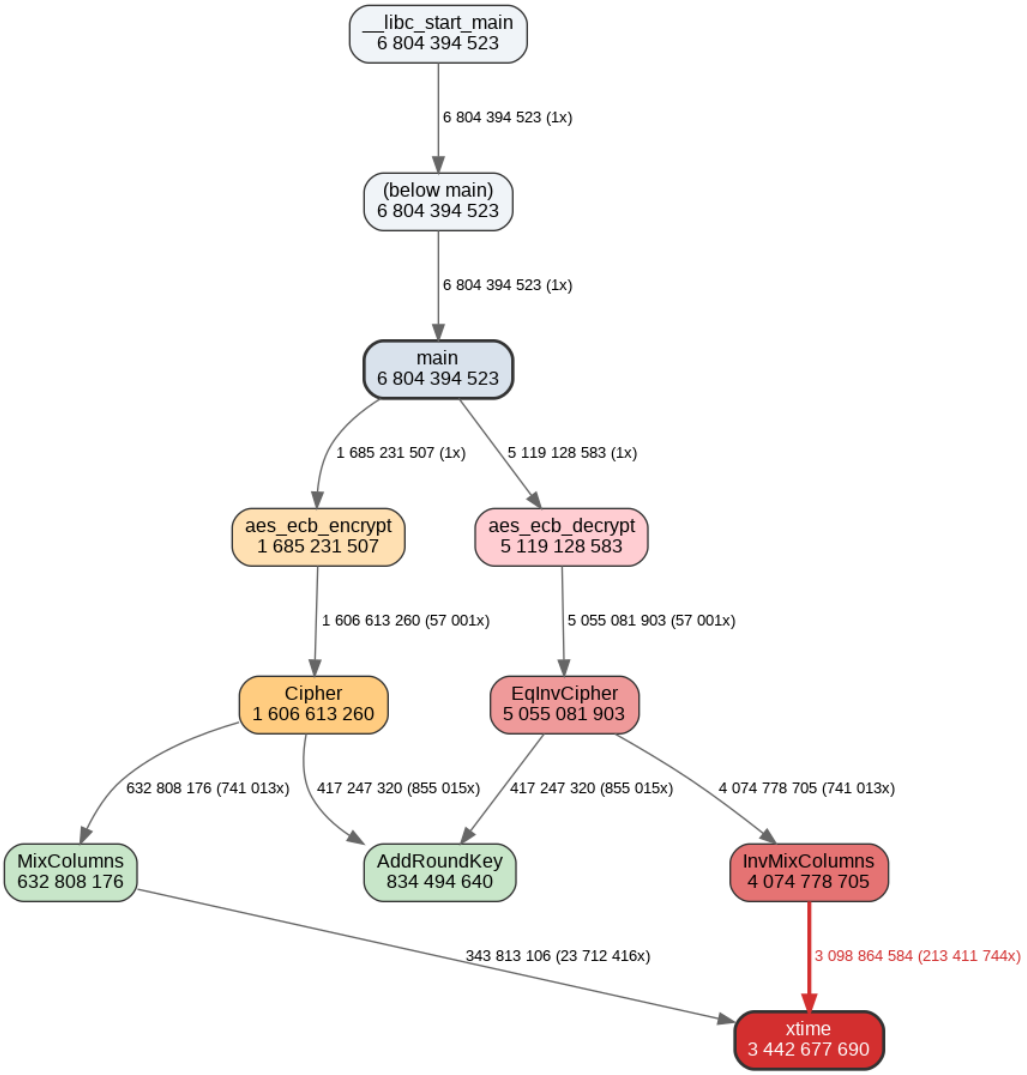


Figure 3 – Profil de performance obtenu sous *Valgrind* et *KCachegrind* pour la variante `EqInvCipher` avec `KeyExpansion` calculée une seule fois.

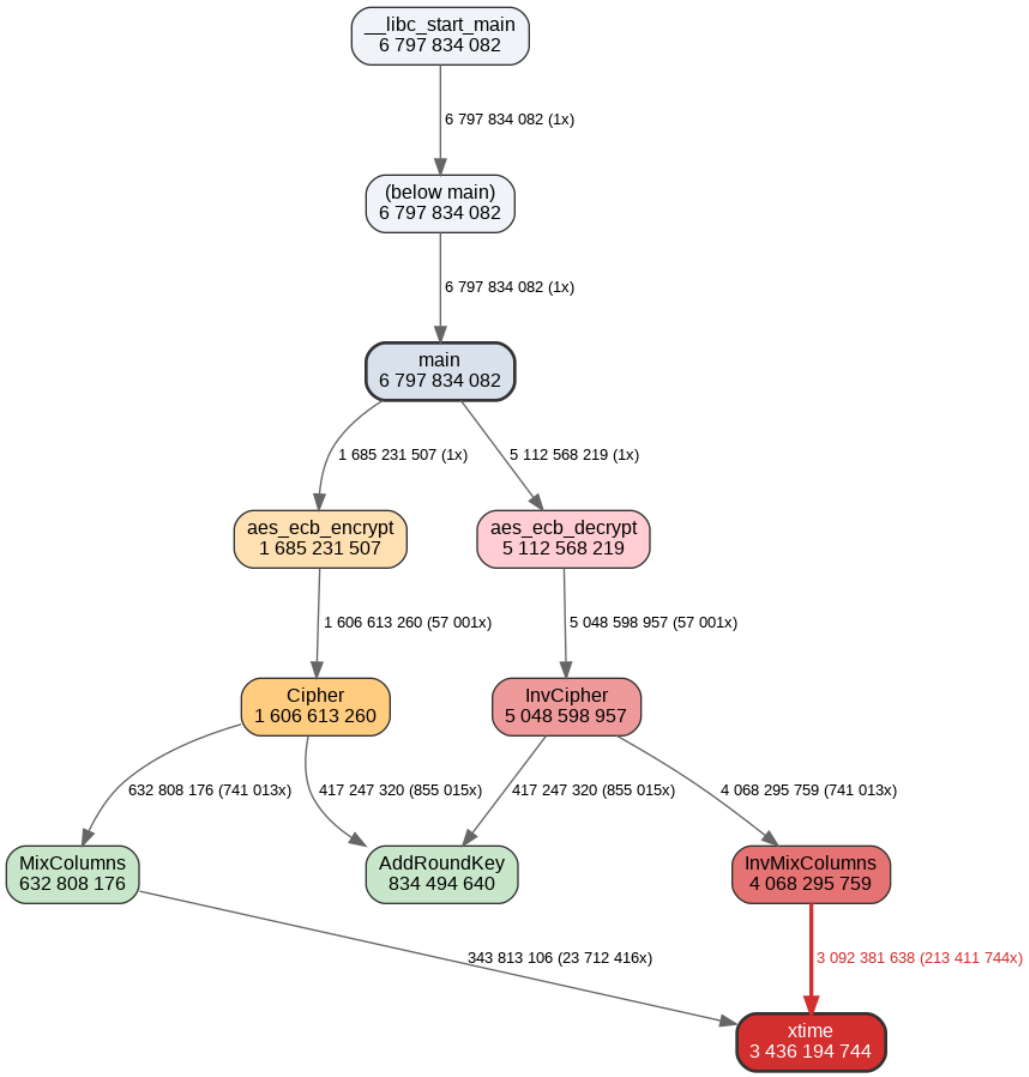


Figure 4 – Profil de performance obtenu sous *Valgrind* et *KCachegrind* pour la variante classique `InvCipher` avec `KeyExpansion` calculée une seule fois.

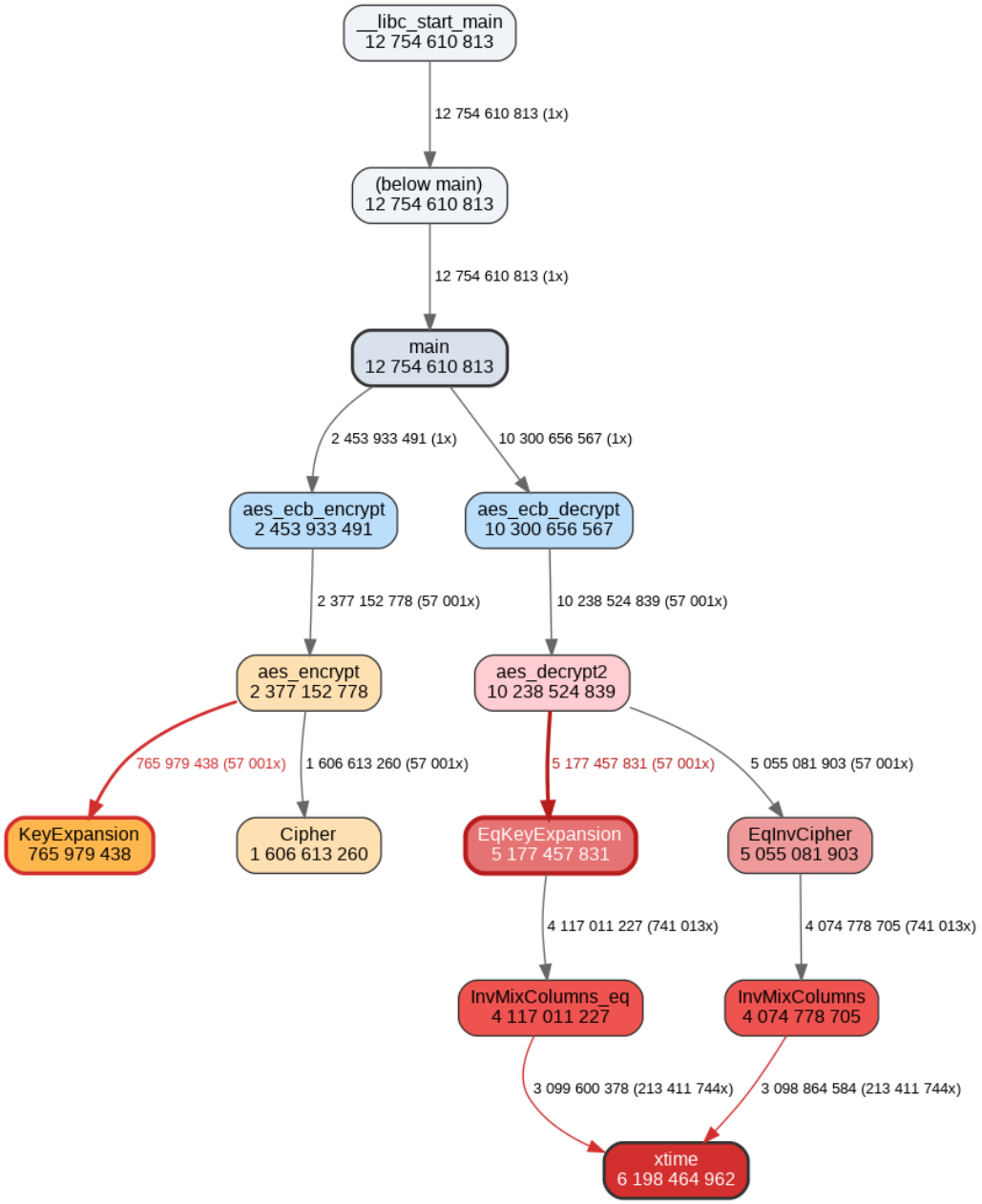


Figure 5 – Profil de performance de la variante `EqInvCipher` lorsque `EqKeyExpansion` est recalculée pour chaque bloc de 16 octets.



Figure 6 – Profil de performance de la variante classique `InvCipher` lorsque `KeyExpansion` est recalculée pour chaque bloc de 16 octets.

L'examen croisé du tableau 7 et des profils d'exécution (3 à 6) conduit à trois constats techniques :

1. **Asymétrie marquée entre chiffrement et déchiffrement :** lorsque la clé étendue est précalculée une seule fois, le déchiffrement (0,745 s avec `EqInvCipher`, 0,735 s avec `InvCipher`) se révèle plus de cinq fois plus lent que le chiffrement (0,143 s). L'inspection de la figure 3 montre que cette asymétrie provient de la fonction `InvMixColumns`, qui engendre à elle seule plus de 213 millions d'appels à la fonction `xtime()`. En effet, alors que le polynôme de `MixColumns` n'utilise que des coefficients de faible degré ($\{01\}, \{02\}, \{03\}$), le polynôme inverse $a^{-1}(X)$ de `InvMixColumns` fait intervenir les coefficients $\{09\} = x^3+1$, $\{0b\} = x^3+x+1$, $\{0d\} = x^3+x^2+1$ et $\{0e\} = x^3+x^2+x$, qui requièrent des compositions d'appels imbriqués jusqu'à $\texttt{xtime}(\texttt{xtime}(\texttt{xtime}(s)))$.

2. **Comparaison entre `EqInvCipher` et `InvCipher` :** dans cette implémentation logicielle en C sur processeur généraliste, `EqInvCipher` n'apporte pas de gain de vitesse par rapport à `InvCipher` (0,745 s contre 0,735 s), car la permutation de l'ordre des appels ne réduit pas le nombre d'évaluations de `xtime()` et ajoute le surcoût initial de `EqKeyExpansion`. L'intérêt architectural de `EqInvCipher` réside principalement dans les implémentations matérielles (ASIC ou FPGA) ou microprogrammées contraintes, où il permet de mutualiser la structure du pipeline de ronde entre le chiffreur et le déchiffreur.
3. **Surcoût critique du recalcul de `KeyExpansion` à chaque bloc :** lorsque l'expansion de clé est réexécutée à l'intérieur de la boucle de traitement des blocs (figures 5 et 6), le débit de chiffrement chute de 6,37 Mo/s à 4,37 Mo/s (+46 % sur le temps d'exécution). Sur le déchiffrement avec `EqInvCipher`, la dégradation est majeure : le temps double presque, passant de 0,745 s à 1,440 s (débit réduit à 633 Ko/s), car `EqKeyExpansion` applique `InvMixColumns_eq` aux sous-clés intermédiaires à chaque bloc de 16 octets. Ces mesures justifient l'architecture retenue dans `ECB.c` et `CBC.c`, où la clé étendue est calculée une unique fois avant la boucle de parcours des blocs.

7.2 Optimisation algorithmique de `MixColumns` et `InvMixColumns`

L'analyse des profils *KCachegrind* (figures 3 à 6) a révélé que la fonction `xtime()` concentrait l'essentiel du temps processeur en raison de recalculs redondants au sein d'une même colonne. Dans une implémentation naïve de `InvMixColumns`, les multiplications par $\{0e\}, \{0b\}, \{0d\}$ et $\{09\}$ recalculent séparément $x \cdot s_i$, $x^2 \cdot s_i$ et $x^3 \cdot s_i$ pour chacune des quatre lignes de la matrice.

Les fonctions `MixColumns`, `InvMixColumns` et `InvMixColumns_eq` ont par conséquent été réécrites dans `AES.c` afin de précalculer et mémoriser dans des variables locales les puissances successives de x pour les quatre octets (s_0, s_1, s_2, s_3) de la colonne courante :

$$\begin{aligned} \texttt{x_s}_i &= \texttt{xtime}(s_i) = x \bullet s_i, \\ \texttt{x_x_s}_i &= \texttt{xtime}(\texttt{x_s}_i) = x^2 \bullet s_i, \\ \texttt{x3_s}_i &= \texttt{xtime}(\texttt{x_x_s}_i) = x^3 \bullet s_i. \end{aligned}$$

Cette factorisation ramène le coût de `InvMixColumns` à exactement $3 \times 4 = 12$ appels à `xtime()` par colonne (soit 48 appels par ronde), toutes les combinaisons linéaires suivantes s'effectuant exclusivement par des opérations XOR (`^`) sur ces variables locales.

Le tableau 8 présente les performances mesurées après cette optimisation algorithmique (avec `KeyExpansion` calculée une seule fois), et la figure 7 montre le nouveau profil d'exécution obtenu sous *KCachegrind*.

Variante évaluée	Opération	Temps d'exécution	Débit mesuré
`InvCipher`	**Chiffrement**	0,132 s	6 908 603 o/s
`InvCipher`	**Déchiffrement**	0,209 s	4 368 806 o/s
`EqInvCipher`	**Chiffrement**	0,132 s	6 890 187 o/s
`EqInvCipher`	**Déchiffrement**	0,211 s	4 327 926 o/s

Table 8 – Performances mesurées après factorisation des appels à `xtime()` dans `MixColumns`, `InvMixColumns` et `InvMixColumns_eq`.

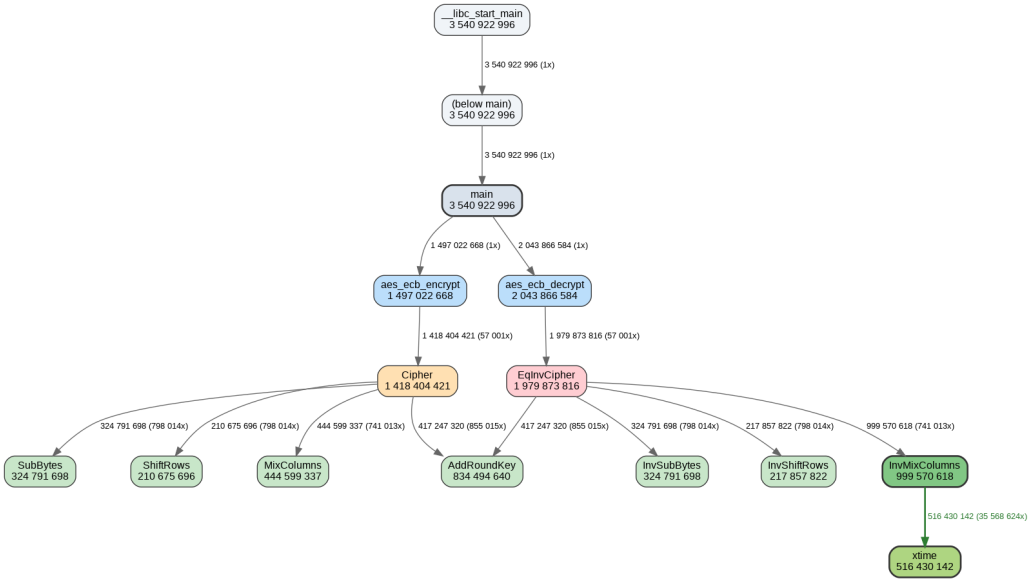


Figure 7 – Profil d'exécution obtenu sous *Valgrind* et *KCachegrind* après optimisation algorithmique des fonctions de mélange de colonnes.

Comme l'atteste la figure 7, le nombre total d'appels à `xtime()` générés par `InvMixColumns` chute de plus de 213 millions à environ 35,5 millions, soit une réduction d'un facteur 6. Le temps de déchiffrement passe ainsi de 0,735 s à 0,209 s, multipliant le débit de déchiffrement par 3,52 (de 1,24 Mo/s à 4,37 Mo/s). En outre, les performances de `InvCipher` (0,209 s) et de `EqInvCipher` (0,211 s) deviennent quasiment indiscernables, démontrant que les écarts observés précédemment résultaient de la redondance des appels à `xtime()` et non d'une différence intrinsèque de complexité entre les deux schémas de déchiffrement.

7.3 Étude des optimisations du compilateur GCC (`-O0` à `-O3`)

Une fois le code source C optimisé sur le plan algorithmique, une campagne de mesures a été menée pour quantifier l'impact des passes d'optimisation du compilateur GCC (`-O0`, `-O1`, `-O2` et `-O3`). Afin de lisser les fluctuations liées à l'ordonnancement du système d'exploitation, chaque configuration a été évaluée en calculant la moyenne sur trois exécutions successives. Le tableau 9 présente les résultats obtenus.

Niveau GCC	Temps Chiff.	Débit Chiffrement	Temps Déchiff.	Débit Déchiffrement
`-O0 (Base)`	∼ 0,132 s	∼ 6 900 000 o/s	∼ 0,209 s	∼ 4 350 000 o/s
`-O1`	0,091 s	10 396 033 o/s	0,107 s	8 514 305 o/s
`-O2`	0,045 s	21 724 127 o/s	0,043 s	21 168 886 o/s
`-O3`	0,036 s	25 854 969 o/s	0,039 s	23 660 786 o/s

Table 9 – Évolution des temps d'exécution et des débits de chiffrement et de déchiffrement en fonction du niveau d'optimisation du compilateur GCC (moyenne sur 3 exécutions).

L'analyse du code machine généré à l'aide du désassembleur `objdump -M intel -dS` permet d'expliquer précisément les gains observés à chaque palier :

- **Palier `-O0` vers `-O1` :** sans optimisation (`-O0`), le compilateur traduit littéralement chaque instruction C en sauvegardant et rechargeant systématiquement les variables locales (`s0`, `x_s0`, compteurs de boucles) sur la pile mémoire (*stack frame*), et émet une instruction `call` complète pour chaque invocation de `xtime()`. L'activation de `-O1` élimine une grande partie de ces accès mémoire superflus en allouant les variables temporaires dans les registres du processeur et en simplifiant les expressions redondantes, ce qui divise presque par deux le temps de déchiffrement (0,107 s contre 0,209 s).
- **Palier `-O1` vers `-O2` :** le niveau `-O2` active l'intégration en ligne (*inlining*) des fonctions courtes, l'ordonnancement avancé des instructions, la propagation des constantes et l'optimisation des boucles bornées. Les débits de chiffrement (21,72 Mo/s) et de déchiffrement (21,17 Mo/s) font plus que doubler par rapport à `-O1` et deviennent pratiquement symétriques.
- **Palier `-O2` vers `-O3` :** le niveau `-O3` applique le déroulement complet des boucles (*loop unrolling*) de taille fixe $Nb = 4$ dans `SubBytes`, `ShiftRows`, `MixColumns` et `AddRoundKey`, supprimant les branchements conditionnels de fin de boucle, et exploite la vectorisation automatique. Le débit atteint alors 25,85 Mo/s en chiffrement (0,036 s, soit un gain d'un facteur 3,7 par rapport à `-O0`) et 23,66 Mo/s en déchiffrement (0,039 s, soit un gain d'un facteur 5,4 par rapport à `-O0`). Le ralentissement relatif de la progression entre `-O2` et `-O3` indique l'approche de l'asymptote de performance atteignable pour cette implémentation logicielle sans instructions matérielles dédiées (telles que AES-NI).

8 Conclusion et bilan technique

Ce projet mené dans le cadre de l'UE PX222 Maths-Info a permis d'explorer l'algorithme de chiffrement AES selon deux paradigmes de programmation complémentaires, mettant en lumière la correspondance profonde entre les structures de l'algèbre abstraite et les mécanismes de la cryptographie symétrique moderne.

Sur le plan de la conception fonctionnelle en Haskell, la structuration du code en quatre couches d'abstraction strictement hiérarchisées, depuis les classes de types `Anneau` et `Corps` jusqu'au moteur `AES.hs`, en passant par l'arithmétique polynomiale générique `Polynome a` et la construction modulaire de GF(2^8), a démontré la pertinence du typage fort pour modéliser fidèlement les mathématiques de la norme FIPS 197. L'extension de la division euclidienne et de l'algorithme d'Euclide étendu à des corps finis quelconques (illustrée sur $\mathbb{Z}/3\mathbb{Z}$), le calcul algébrique dynamique de la S-Box (par inversion multiplicative dans GF(2^8) suivie d'une transformation affine modulo $X^8 + 1$) et l'expression concise de `mixColumns` dans l'anneau quotient $\mathrm{GF}(2^8)[X]/(X^4 + 1)$ témoignent d'une architecture modulaire, exempte de dépendances circulaires et intégralement validée sur les vecteurs officiels des annexes A, B et C du NIST pour les variantes AES-128, AES-192 et AES-256.

Sur le plan expérimental et système en langage C, la mise en œuvre des modes opératoires ECB et CBC avec bourrage PKCS#7 sur l'image `Tux.bmp` a illustré concrètement la nécessité du chaînage inter-blocs pour masquer les redondances structurelles d'un fichier clair, faisant passer l'entropie de Shannon de 3,938782 bits/octet (6,994900 en ECB) à 7,999785 bits/octet en CBC, tout en rappelant qu'une entropie proche de 8 constitue une condition nécessaire mais non suffisante de sécurité cryptographique. Enfin, la démarche d'ingénierie des performances conduite avec *Valgrind*, *KCachegrind* et `objdump` a mis en évidence une leçon méthodologique essentielle : l'identification par profilage du goulot d'étranglement lié aux appels redondants à `xtime()` (réduits de plus de 213 millions à environ 35,5 millions dans `InvMixColumns`), conjuguée au précalcul unique de `KeyExpansion` et aux optimisations `-O3` du compilateur GCC, a permis d'atteindre des débits de 25,85 Mo/s en chiffrement et de 23,66 Mo/s en déchiffrement, démontrant que l'analyse fine des données d'exécution prime toujours sur les conjectures d'optimisation *a priori*.

Références

[1] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, FIPS PUB 197, 2001.

[2] Joan Daemen et Vincent Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer, 2002.

[3] Simon Marlow (éd.), *Haskell 2010 Language Report*, 2010.

[4] Rudolf Lidl et Harald Niederreiter, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, 1986.