Analyse statistique visuelle

> *« The field of cryptography will perhaps be the most rewarding. There is a remarkably close parallel between the problems of the physicist and those of the cryptographer. The system on which a message is enciphered corresponds to the laws of the universe, the intercepted messages to the evidence available, the keys for a day or a message to important constants which have to be determined. The correspondence is very close, but the subject matter of cryptography is very easily dealt with by discrete machinery, physics not so easily. »*
>
> Alan Turing – Intelligent Machinery

Dans le domaine de la sécurité des systèmes d'information, la visualisation est couramment utilisée pour différentes tâches comme l'analyse de logs [197], la détection d'attaques [149], l'analyse de binaires [79] et l'ingénierie inverse [55, 54], mais aujourd'hui, il n'existe pas de façon simple d'analyser et de différencier des données aléatoires. Cependant, les systèmes d'exploitation ou les protocoles cryptographiques utilisent constamment la génération d'aléa, par exemple, pour générer un numéro de séquence TCP ou pour générer une clef de chiffrement aléatoire pour le wi-fi ou le Web.

5.1 Algorithme cryptographique et aléa

Le Graal de tout algorithme cryptographique est d'obtenir, à chaque étape interne et à l'issue du processus de chiffrement, une séquence d'apparence la plus proche possible de l'aléa parfait. En effet, la sécurité d'un algorithme cryptographique dépend de sa capacité à générer des quantités imprévisibles.
En partant du principe que l'aléa parfait n'est qu'une vision philosophique et que, dans les faits, la perfection de l'aléa est tributaire des tests statistiques qui lui ont été appliqués [97], nous pouvons dire que l'aléa cryptographique doit être aléatoire dans le sens où la probabilité d'une valeur particulière choisie doit être suffisamment faible pour empêcher un adversaire de gagner l'avantage grâce à l'optimisation d'une stratégie de recherche basée sur cette probabilité [142].

Le moyen pour arriver à réaliser ce paradigme est de concevoir les algorithmes cryptographiques comme des générateurs de bits pseudo-aléatoires (**Pseudo-random Bit Generator** ou PRBG en anglais).

5.1.1 Générateur de bits pseudo-aléatoire

Définition 5.1 (*Générateur de bits aléatoire*)**.** Un générateur de bits aléatoire est un équipement ou un algorithme qui produit une séquence de bits statistiquement indépendants et non biaisés [142].

Certains équipements matériels génèrent de l'aléa à partir du temps écoulé entre l'émission de particules durant la phase de décroissance radioactive ou encore à partir du bruit thermique émis par une résistance ou une diode à semi-conducteurs. De même, pour générer de l'aléa, certains logiciels utilisent des algorithmes associant diverses sources comme le temps écoulé entre deux frappes au clavier, le mouvement de la souris ou le contenu des buffers d'entrée/sortie.

Définition 5.2 (*Générateur de bits pseudo-aléatoire*)**.** Un générateur de bits pseudo-aléatoire (PRBG) est un algorithme déterministe qui, à partir d'une séquence de bits réellement aléatoire de longueur k, appelée graine, produit une séquence de bits de longueur $l \gg k$ qui semble aléatoire. La séquence initiale est appelée la graine tandis que la séquence produite par le PRBG est appelée séquence de bits pseudo-aléatoire.

Générer des nombres aléatoires est un problème très difficile pour les ordinateurs parce qu'ils sont déterministes et, comme le dit John von Neumann *« Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin »* [204]. Ainsi, la séquence produite par un PRBG n'est pas réellement aléatoire. Plus précisément, le nombre de séquences possibles produites par le PRBG est, au plus, une petite fraction, à savoir $\frac{2^k}{2^l}$ [142], de toutes les séquences binaires possibles de longueur l. L'objectif, est de prendre une petite séquence réellement aléatoire et de l'étendre à une séquence de longueur beaucoup plus grande, de telle sorte que l'attaquant ne puisse pas facilement faire la distinction entre les séquences de sortie du PRBG et des séquences réellement aléatoires de même longueur.
Pour qu'un algorithme de PRBG soit cryptographiquement sûr, trois principales règles doivent être respectées [142] :

1. la longueur k de la graine doit être de taille suffisante. En fait, k doit être telle, que la recherche exhaustive sur l'ensemble des 2^k éléments de l'espace des graines soit calculatoirement difficile pour l'attaquant ;
2. la séquence produite par un PRBG doit être statistiquement proche d'une séquence réellement aléatoire, ou, plus précisément, approximée par une séquence de variables binaires, indépendantes et identiquement distribuées. Nous disons alors qu'un PRBG passe tous les tests statistiques en temps polynomial si aucun algorithme polynomial ne peut correctement faire la distinction entre une séquence produite par le PRBG et une séquence réellement aléatoire de même longueur avec une probabilité $p \gg \frac{1}{2}$;
3. les bits produits ne doivent pas être prédictibles, à partir d'une séquence partielle déjà connue, pour un attaquant ayant des ressources de calcul

limitées. Un PRBG respecte cette règle, dite du "bit suivant", si, à partir des premiers l bits d'une séquence s produite par le PRBG, aucun algorithme polynomial n'est capable de prédire le bit $(l+1)$ de s avec une probabilité $p \gg \frac{1}{2}$.

5.1.2 Représentation de PRBG

Afin de mieux appréhender le résultat obtenu à partir de différents algorithmes de PRBG, nous allons représenter les séquences obtenues dans un environnement en deux et trois dimensions simultanément.

5.1.2.1 Représentation en 2 dimensions

Un PRBG est assimilable à un système non-linéaire générant une série chronologique de données. Si nous voulons représenter une telle série dans un environnement en deux dimensions, une première approche pourrait consister à parcourir linéairement tous les points du plan en assignant à chaque point une couleur correspondant à une entrée de la série. Cette idée semble bonne mais elle a l'inconvénient de ne pas représenter la réalité de la série.
En effet, si nous prenons un plan délimité par un rectangle de largeur x et de hauteur y avec $x \times y = |n|$, $|n|$ représentant le cardinal des éléments de la série n, alors le point de coordonnées (i, j), représentant l'élément $n(t), t < |n|$ de la série, a comme voisins les points $(i-1, j)$ et $(i+1, j)$ représentant respectivement les éléments $n(t-1)$ et $n(t+1)$ de la série, mais a également comme voisins les points $(i, j-1)$ et $(i, j+1)$ représentant les éléments $n(t-x)$ et $n(t+x)$ de la série. Ainsi, les points sur une même ligne correspondent à des éléments qui se suivent dans la série mais il n'y a pas de lien exploitable entre différentes lignes. À titre d'exemple, nous pouvons voir que sur la figure 5.1 page 91, le point d'index 15 a comme voisins les points d'index 14 et 16 mais également les points d'index 8 et 20 qui ne lui sont pas voisins dans la séquence.

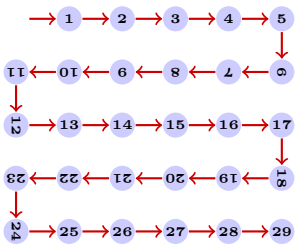


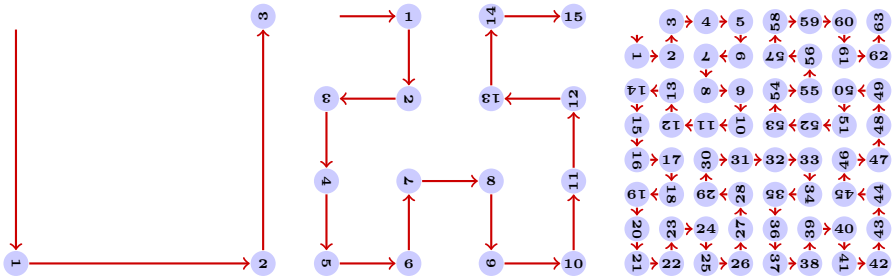
FIGURE 5.1 – Courbe standard

Donc, pour obtenir une meilleure représentation de notre série dans un espace à deux dimensions nous avons besoin d'un algorithme plus pertinent. Une courbe de remplissage est une fonction injective continue qui fait correspondre un intervalle compact à un hypercube n-dimensionnel [110]. Les courbes de remplissage ont été découvertes en 1890 par le mathématicien Giuseppe Peano [160]. Pour réaliser notre objectif nous utilisons une courbe de remplissage spécifique proposée par David Hilbert [113] peu de temps après la découverte de Peano.
Une courbe de Hilbert est une courbe de remplissage qui réalise la projection d'un intervalle à une dimension dans un espace à deux dimensions. Sa construc-

tion repose sur la répétition d'un schéma simple : les trois premiers côtés d'un carré. À chaque étape le carré est tourné, réduit et répété jusqu'à obtenir une courbe qui remplit le plan. En fait, une courbe de Hilbert peut être vue comme un système de Lindenmayer [210], connu également sous le nom de L-system. Un L-system est un système de réécriture de séquence de caractères qui peut être utilisé pour générer des fractales de dimensions comprises entre un et deux. Pour la courbe de Hilbert les règles du L-system sont :

$$L = +RF - LFL - FR+$$
$$R = -LF + RFR + FL-$$

avec l'alphabet constitué des lettres L et R, F signifiant dessiner vers l'avant, $-$ signifiant tourner à gauche de 90 degrés et $+$ signifiant tourner à droite de 90 degrés.
Les premières itérations d'une courbe de Hilbert sont présentées dans la figure 5.2. Comme nous pouvons le voir sur les figures 5.2a, 5.2b et 5.2c, chaque point a un index correspondant à l'index de chaque entrée de la série. Nous constatons alors que, contrairement à notre première approche décrite ci-dessus, l'utilisation d'une courbe de Hilbert nous permet de préserver la localité des données. Nous entendons par là, que le voisinage des points dans la série est conservé dans l'espace à deux dimensions.



(a) Ordre 1 (b) Ordre 2 (c) Ordre 3

FIGURE 5.2 – Premières itérations de la courbe de Hilbert

Ainsi, au travers de ce mode de présentation en deux dimensions, nous avons une première approche pour représenter un algorithme de PRBG.

5.1.2.2 Représentation en 3 dimensions

Après l'approche en deux dimensions nous allons essayer de représenter notre séquence dans un environnement en trois dimensions.
Un des moyens les plus couramment utilisé pour analyser une série de ce type est de reconstruire son espace des phases en utilisant la méthode des délais [119]. L'espace des phases est un espace à n dimensions qui décrit complètement l'état d'un système à n variables. À titre d'exemple, l'espace des phases décrivant l'atterrissage d'une fusée est un espace à deux dimensions. La première dimension est la vitesse de la fusée et la deuxième dimension est sa distance au sol. L'espace des phases est alors un graphe représentant en abscisse la vitesse et en ordonnée la distance au sol. Ainsi, pour que la fusée atterrisse sans encombre, il

faut que la courbe décrivant sa progression, dans son espace des phases, tende vers zéro (voir figure 5.3 page 93).

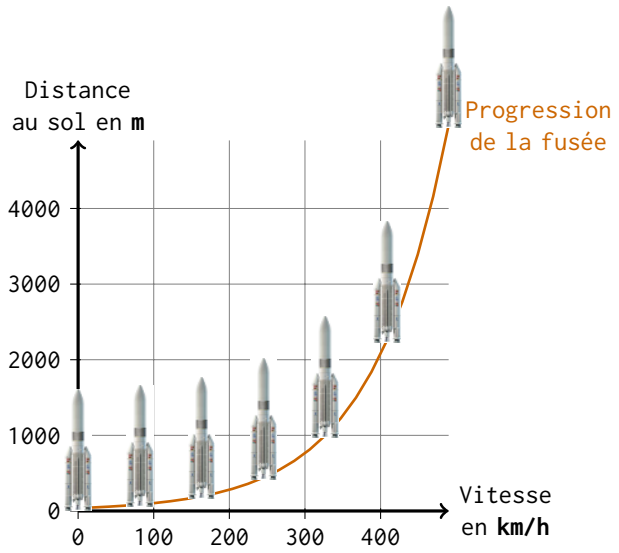


FIGURE 5.3 – Espace des phases de la trajectoire d'une fusée

Ici, notre objectif est donc de représenter en trois dimensions une séquence à une dimension. La méthode des délais [159] permet de reconstruire les dimensions manquantes en utilisant les valeurs précédentes comme coordonnées supplémentaires. Pour cela, au lieu d'utiliser les valeurs brutes retournées par la fonction, nous calculons, pour chaque coordonnée, la différence de deux valeurs successives. Cela nous permet de générer un résultat plus utile pour montrer la dynamique de la fonction. Ainsi, si $s[t]$ est la séquence fournie par un PRBG en fonction du temps t, alors les coordonnées x, y, z d'un point dans notre environnement sont calculées à partir des équations suivantes :

$$x[t] = s[t-2] - s[t-3]$$
$$y[t] = s[t-1] - s[t-2]$$
$$z[t] = s[t] - s[t-1]$$

Ensuite, en représentant la séquence de points ainsi obtenue dans un environnement en trois dimensions nous obtenons une forme spécifique à la fonction de PRBG donnée. Cette forme, appelée *attracteur*, révèle la nature complexe des dépendances entre les différents éléments de la séquence générés par l'algorithme étudié [216, 217].
Prenons un exemple concret : la suite de nombres suivante est issue de l'algorithme de PRBG qui génère les numéros de séquences de session TCP du système d'exploitation GNU/Linux RedHat dans sa version 7.3.

```
3281499104, 3271545868, 3287443610, 3238749981, 3274168813, 3302234066,
3229771300, 3287970591, 3295595222, 3298841199, 3292774952, 3294591612,
3294540537, 3294046036, 3296969037, 3293746299, 3300112100, 3292483752,
3235813772, 3298333679, 3273849495, 3293225350, 3295916141, 3299559674,
3295896492, 3303667282, 3301180722, 3290619488, 3301904507, 3286172964
```

Au premier abord, il est difficile de déterminer s'il existe un lien entre les différents éléments de cette suite de nombres. Par contre, la figure 5.4 page 94 permet de formaliser ce lien en dévoilant une forme qui est caractéristique du PRBG utilisé par le noyau Linux 2.4.18 utilisé par cette distribution de Linux.

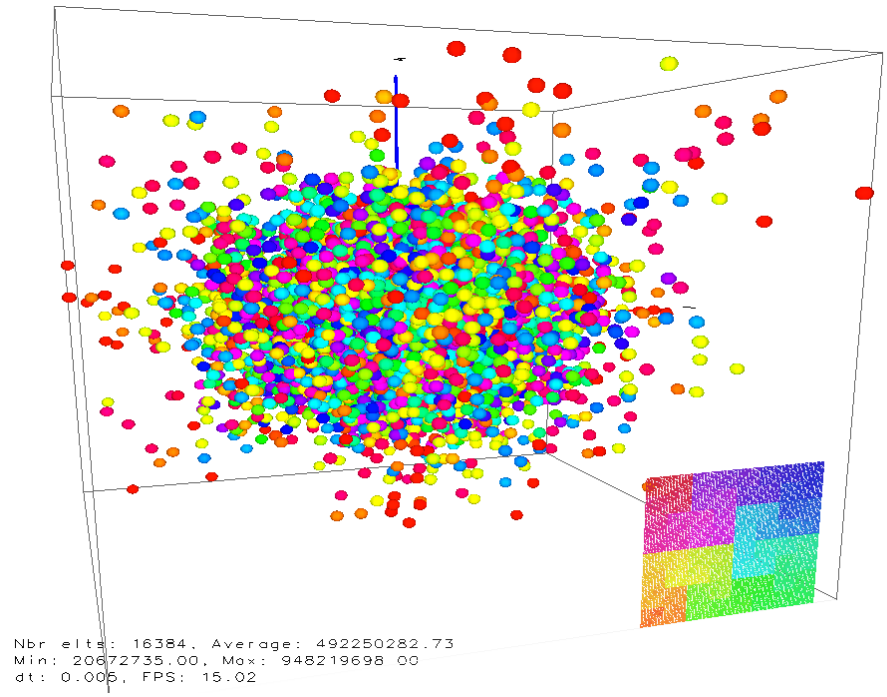


FIGURE 5.4 – Attracteur du PRBG de Redhat 7.3 - vue 45°

Grâce à ce mode de présentation, il nous est maintenant plus facile d'identifier un algorithme de PRBG, étant donné qu'un même algorithme donnera toujours le même attracteur.
Pour la suite de ce chapitre, nous avons développé un ensemble de programmes de génération et de visualisation en trois dimensions de différents types de PRBG. Ces programmes sont disponibles sur Internet [81, PRBG-3D].

5.1.3 Exemples de PRBG

Nous avons vu plus haut qu'un algorithme cryptographique doit être conçu comme un générateur de bits pseudo-aléatoire. Nous allons maintenant étudier quelques algorithmes de PRBG. Nous finirons par la présentation de l'attracteur du RC4 et de l'*Advanced Encryption Standard*.

5.1.3.1 Véritable aléa

Avant de débuter notre comparaison de PRBG, il nous faut un référentiel, c'est-à-dire la représentation de l'attracteur d'un véritable aléa. Comme un ordinateur est une machine déterministe, il ne lui est pas possible de produire un véritable aléa. Seuls des équipements matériels s'appuyant sur des éléments aléatoires physiques peuvent fournir ce type d'aléa.
Le site Internet www.random.org fournit la possibilité de générer des séquences de véritable aléa. Il utilise le bruit atmosphérique pour produire cet aléa. Ce site est issu d'un projet scientifique du docteur Mads Haahr de la "*School of Computer Science and Statistics*" du Trinity College de Dublin [109]. Il est utilisé pour des jeux en lignes, pour générer l'aléa de jeux de loterie, pour des projets scientifiques... Nous avons généré une séquence de 10 000 entiers aléatoires à partir de ce site. Les premières entrées de cette séquence sont les suivantes :

```
72329, 95447, 11130, 52803, 25986, 58390, 84305, 98618, 54545, 64850,
27412, 15977, 13214, 30421, 91625, 48878, 35783, 58844, 16061, 74799,
99777, 87273, 61979, 77926, 66628, 56546, 19300, 34809, 11633, 86476
```

Et la représentation en 3 dimensions de l'attracteur correspondant est montrée dans la figure 5.5 page 95.

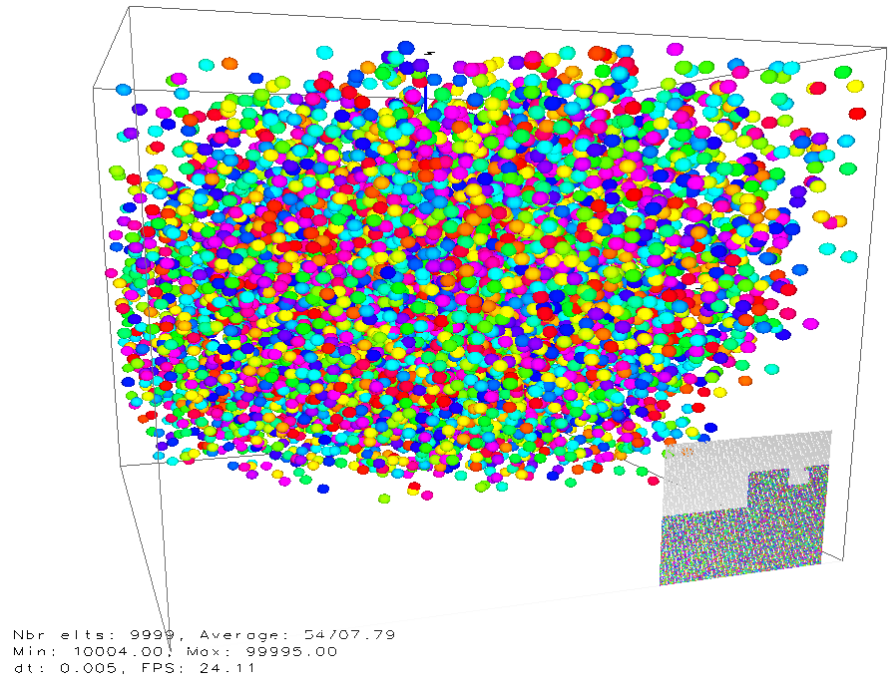


FIGURE 5.5 – Attracteur d'un aléa véritable - vue 45°

La figure 5.5 page 95 montre un environnement en trois dimensions, matérialisé par un cube et les trois axes x en rouge, y en vert et z en bleu. La répartition des points dans cet espace forme un nuage homogène et couvre uniformément un volume sphérique selon les trois axes. Aucun schéma spécifique ne se dégage. Les valeurs fournies sont comprises entre 0 et 99995 avec une valeur moyenne

```
1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3,
2, 3, 8, 4, 6, 2, 6, 4, 3, 3, 8, 3, 2, 7, 9,
5, 0, 2, 8, 8, 4, 1, 9, 7, 1, 6, 9, 3, 9, 9,
3, 7, 5, 1, 0, 5, 8, 2, 0, 9, 7, 4, 9, 4, 4,
5, 9, 2, 3, 0, 7, 8, 1, 6, 4, 0, 6, 2, 8, 6,
2, 0, 8, 9, 9, 8, 6, 2, 8, 0, 3, 4, 8, 2, 5,
        3, 4, 2, 1, 1, 7, 0, 6, 7, 9
```

FIGURE 5.6 – Les 100 premières décimales de Pi

de 54707.79. Dans le coin droit, nous avons la représentation de l'ensemble des données à l'aide de la courbe de Hilbert.
De plus, le programme que nous utilisons, attribue une couleur spécifique à chaque point : l'ensemble des couleurs de la palette graphique est réparti sur l'ensemble des points de façon chronologique. Ainsi, le premier point sur la liste reçoit la première couleur de la palette, le deuxième point reçoit la deuxième couleur et ainsi de suite jusqu'au dernier point. Ce principe nous permet de rajouter une quatrième dimension à notre graphe : le temps.
Sur notre courbe la répartition des couleurs semble complètement aléatoire.

5.1.3.2 Les décimales de Pi

Le nombre Pi est une constante mathématique définie par le ratio de la circonférence d'un cercle et de son diamètre. Pi est communément approximé par 3.14159265. Étant un nombre irrationnel, Pi ne peut pas être exprimé exactement sous la forme d'une fraction. Enfin, les décimales de Pi semblent être distribuées de façon aléatoire, cependant aucune preuve de cet état de fait n'a été découverte [212].
Si nous prenons les 100 premières décimales de Pi, nous obtenons une séquence d'entiers décrite dans la figure 5.6 page 96. La représentation dans un environnement en trois dimensions de cette séquence donne l'attracteur de la figure 5.7 page 97.
Dans la figure 5.7, nous pouvons voir que la courbe de Hilbert montre une distribution des couleurs qui semble aléatoire. En revanche, la représentation en trois dimensions montre un nuage de points proche de l'aléa véritable mais avec des alignements ordonnés. Il semble que les décimales de Pi ne sont pas véritablement aléatoire.
En fait, cet effet d'alignement ordonné est dû à la limite des décimales de Pi. En effet ces dernières sont comprises entre 0 et 9 et les différentes combinaisons possibles de ces décimales dans les soustractions que nous faisons pour calculer les cordonnées x, y et z sont limitées. En les regroupant par 4, nous obtenons une répartitions des résultats qui s'étage entre 0 et 9999. Le résultat obtenu est donc plus pertinent. La figure 5.8 montre alors un nuage parfaitement aléatoire. On a donc une première approche de la confirmation de l'hypothèse de l'aléa des décimales du nombre Pi.

5.1.3.3 Générateur de congruence linéaire

Le générateur de congruence linéaire produit une séquence pseudo-aléatoire de nombres entiers x_1, x_2, x_3... en fonction de la récurrence linéaire suivante [142, 124] :

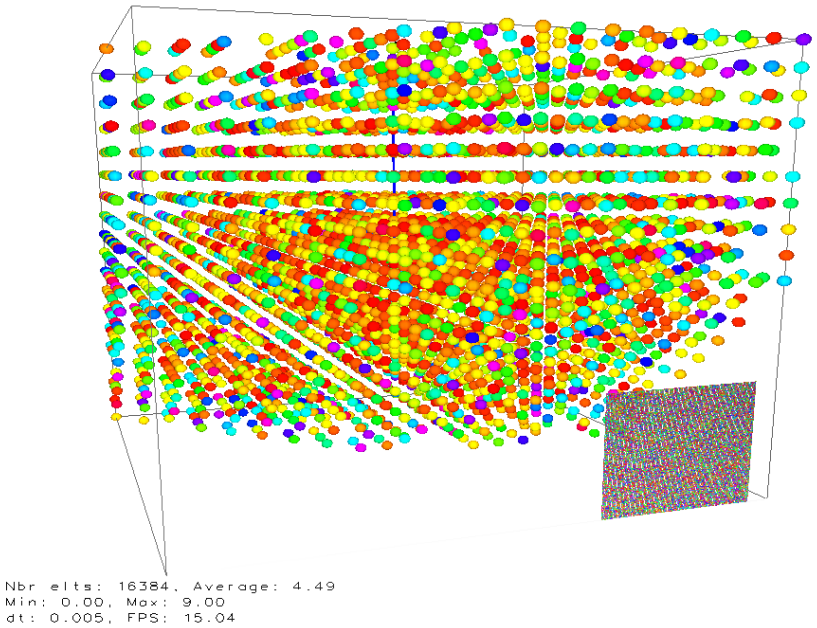


FIGURE 5.7 – Attracteur des décimales de Pi - vue 45°

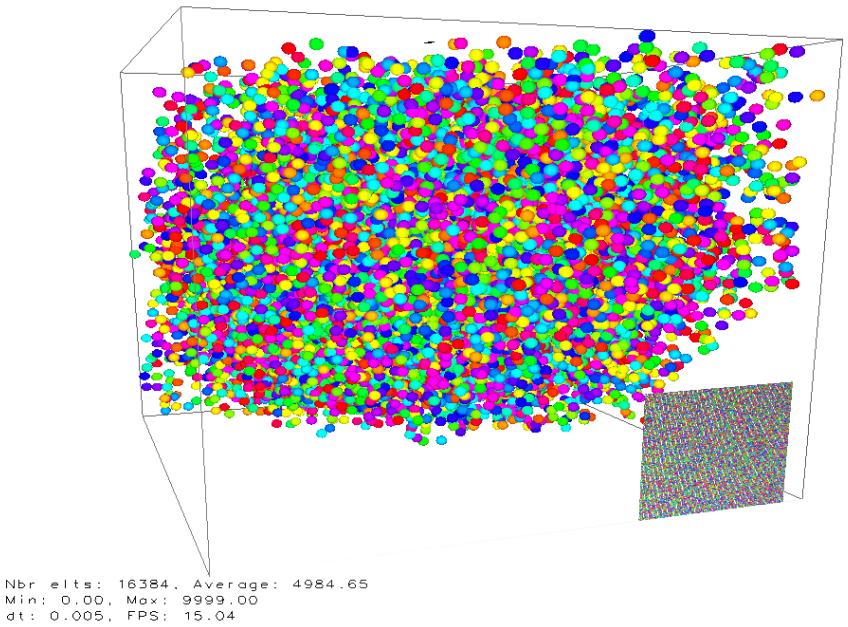


FIGURE 5.8 – Attracteur des décimales de Pi (regroupement par 4) - vue 45°

$$x_{n+1} = ax_n + b \pmod{m} \quad | \quad n \geq 0$$

Les entiers a, b et m sont les paramètres qui caractérisent le générateur et x_0 est la graine. Un exemple de séquence produite avec $a = 5$, $b = 3$ et $m = 4096$ est donné ci-dessous :

```
772, 3863, 2934, 2385, 3736, 2299, 3306, 149, 748, 3743, 2334, 3481,
1024, 1027, 1042, 1117, 1492, 3367, 454, 2273, 3176, 3595, 1594, 3877,
3004, 2735, 1390, 2857, 2000, 1811, 866, 237, 1188, 1847, 1046, 1137,
1592, 3867, 2954
```

Le code en langage C permettant de coder cette fonction est donné dans la figure 5.9 page 98 et l'attracteur correspondant est présenté dans les figures 5.10 page 98 et 5.11 page 99.

```c
double generateLinearCongruence(void) {
    static double xn = 1;
    double value = xn;
    xn = fmod((5 * xn + 3), 4096); //equivalent to (5 * xn + 3) % 4096
    return value;
}
```

FIGURE 5.9 – Code C pour le générateur de congruence linéaire

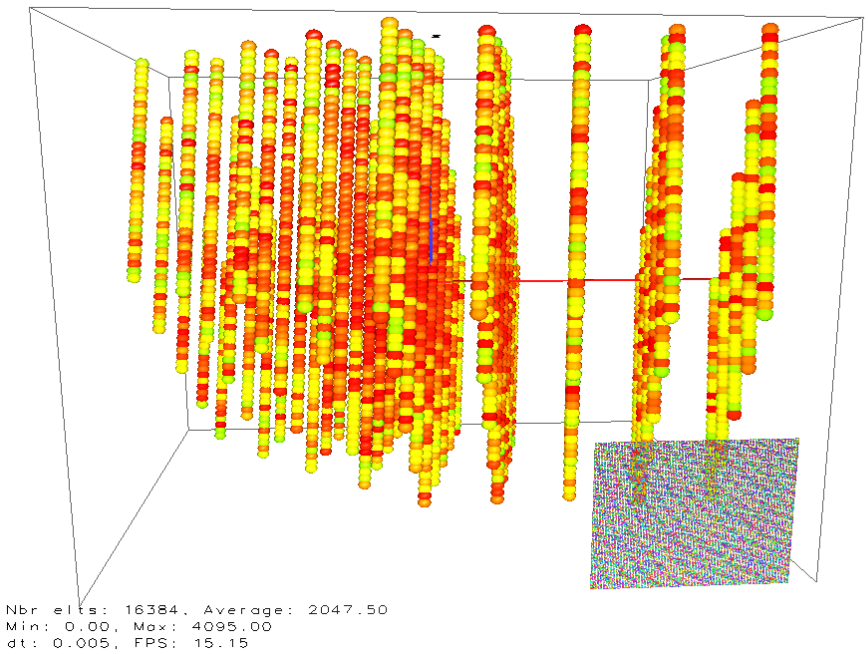


FIGURE 5.10 – Attracteur du générateur de congruence linéaire - axe $y \rightarrow y'$

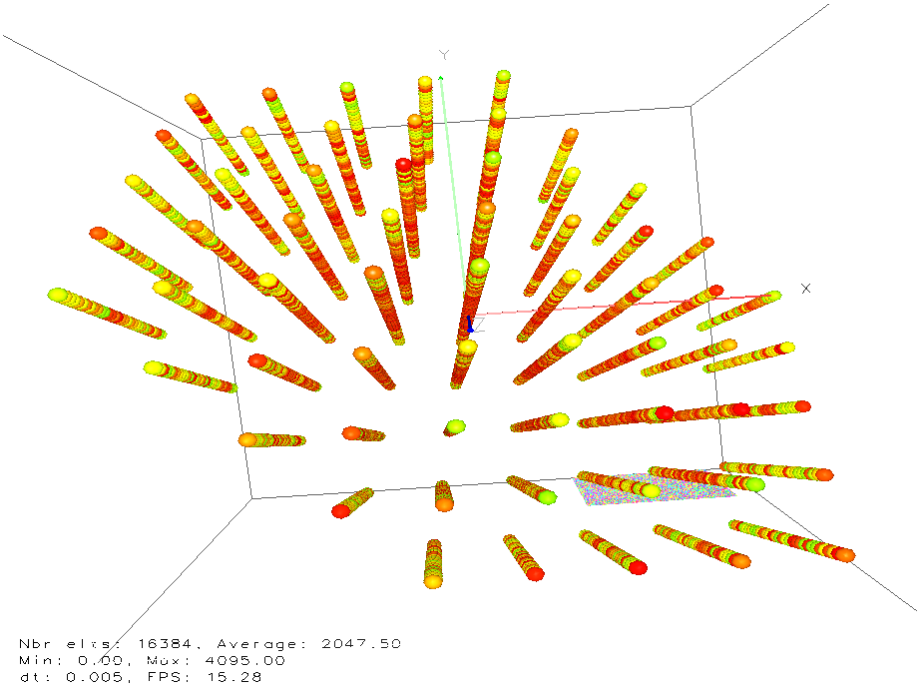


FIGURE 5.11 – Attracteur du générateur de congruence linéaire - axe $z \rightarrow z'$

La taille de l'échantillon est de 16384, la valeur minimale est 0 et la valeur maximale 4095, la valeur moyenne est 2047.50.
La distribution des couleurs dans la courbe de Hilbert semble aléatoire.
Mais, contrairement au générateur de véritable aléa que nous avons pris comme référence, la répartition des points dans les figures 5.10 et 5.11 présente un schéma spécifique sous forme de lignes alignées sur plusieurs plans. Ces lignes correspondent à la période de l'algorithme.
Nous constatons enfin, que seules les couleurs à dominante rouge apparaissent, cela signifie que seule la fin du spectre est visible. Les premiers points sont recouverts par les derniers, l'algorithme fournit donc plusieurs fois les mêmes valeurs.
Ce type de générateur est prédictible et ne convient pas pour un usage cryptographique. En effet, à partir d'une séquence partielle, et sans connaissance des paramètres a, b et m, il est possible de reconstruire le reste de la séquence. De plus, dans ce cas, les deux représentations – en deux et trois dimensions – nous permettent d'affiner la propriété de l'aléa de cet algorithme.

5.1.3.4 Générateur à récurrence non linéaire

Pour notre exemple suivant, nous utilisons un générateur à récurrence non linéaire. Ce dernier produit une séquence pseudo-aléatoire de nombres entiers x_1, x_2, x_3... en fonction de la relation de récurrence suivante [211] :

$$x_{n+1} = \lambda x_n(1 - x_n) \quad | \quad n \geq 0$$

Cette suite dite "logistique" conduit, si $\lambda > 3,56995$, à une suite chaotique. La

suite logistique est utilisée pour modéliser la taille d'une population biologique au fil des générations [140]. Un exemple de séquence produite est donné ci-dessous :

```
451098855224, 451068728783, 451038599037, 451008465983, 450978329622,
450948189954, 450918046977, 450887900692, 450857751097, 450827598193,
450797441977, 450767282451, 450737119614, 450706953464, 450676784002,
450646611227, 450616435137, 450586255734, 450556073016, 450525886982,
450495697633, 450465504967, 450435308985, 450405109684, 450374907066,
450344701129, 450314491873, 450284279297, 450254063401, 450223844184,
450193621646, 450163395785, 450133166603, 450102934097, 450072698268
```

Le code en langage C permettant de coder cette fonction est donné dans la figure 5.12 page 100 et l'attracteur correspondant est présenté dans la figure 5.13 page 101.

```c
double generateLogisticMap(void) {
    static double xn = 0.7364738523; // [0 .. 1]
    static double lambda = 3.8; // > 3,56995
    double value = xn;
    xn = (lambda * xn) * (1 - xn);
    return value;
}
```

FIGURE 5.12 – Code C pour le générateur à récurrence non linéaire

La figure 5.13 page 101 montre également un schéma spécifique ce qui en fait un PRBG non sûr pour une application cryptographique. Cependant, contrairement au générateur de congruence linéaire, nous n'avons plus d'apparition de schéma périodique. Les données générées ne sont pas aléatoires durant les premières itérations (couleur verte) puis semblent le devenir davantage avec l'augmentation du nombre d'itérations.
Dans ce cas, l'étude de la courbe de Hilbert est intéressante parce que la distribution des couleurs semble aléatoire au début mais devient rapidement uniforme à la fin de liste (couleur identique).
Dans notre exemple, la taille de l'échantillon est de 16384, la valeur minimale est 184264892666.96 et la valeur maximale 948850856936.47, la moyenne de l'échantillon est de 631015962902.33.

5.1.3.5 Générateur Blum-Blum-Shub

Le générateur de bits pseudo-aléatoire Blum-Blum-Shub [23] est un PRBG calculatoirement sûr tant que la factorisation de grands nombres composés reste un problème calculatoirement difficile. Ce générateur produit une séquence de bits pseudo-aléatoire selon l'algorithme suivant :

- générer deux grands nombres premiers de Blum p et q et calculer $n = pq$. Un nombre premier de Blum est un nombre premier congruent à 3 modulo 4 ;
- choisir une graine s dans l'intervalle $[1, n-1]$ tel que $\text{pgcd}(s, n) = 1$;
- calculer $x_0 = s^2 \pmod{n}$;
- la séquence est définie comme $x_{i+1} = x_i^2 \pmod{n}$;

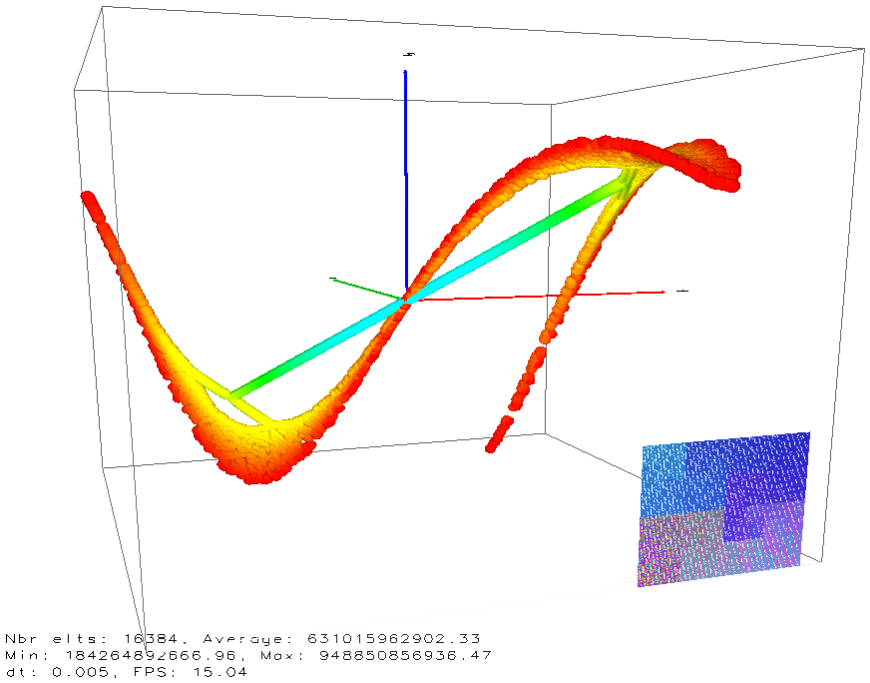


FIGURE 5.13 – Attracteur du générateur à récurrence non linéaire - axe $y \rightarrow y'$

— si x_i est pair alors $z_i = 0$ et si x_i est impair alors $z_i = 1$;
— la sortie de la séquence est z_1, z_2, z_3,...

Le code en langage C permettant de coder cette fonction est donné dans la figure 5.14 page 102 et l'attracteur correspondant est présenté dans la figure 5.15 page 102. Pour faciliter la mise en œuvre de notre présentation, nous n'avons pas appliqué la dernière étape consistant à récupérer le bit de poids faible. Nous constatons que l'attracteur obtenu se répartit sur les trois axes sous la forme d'un nuage homogène. Nous retrouvons la même forme que dans le cas de l'aléa véritable. En fait, le générateur Blum-Blum-Shub est cryptographiquement sûr en supposant l'insolubilité de la résiduosité quadratique [22]. Dans notre exemple, la taille de l'échantillon est de 16384, la valeur minimale est 56782881.0 et la valeur maximale est 512462853845.0, la moyenne de l'échantillon est de 257668663192.64.

5.1.4 Algorithmes cryptographiques

5.1.4.1 Attracteur de RC4

Pour calculer l'attracteur de l'algorithme de RC4, nous utilisons une séquence d'entiers $n = 2^{105} + i$ avec $i = [1 \cdots 60\ 000]$ que nous convertissons en blocs de 128 bits. Ensuite, nous chiffrons chacun de ces blocs avec la même clef de chiffrement de 64 bits :

$$k = 0101010101010101 \tag{5.1}$$

```c
double generateBlumBlumShub(void) {
    // https://en.wikipedia.org/wiki/Blum_Blum_Shub
    // x0 = (s*s) mod(M) avec M=p.q et p, q nombre de Blum
    // p = 869393, q = 589497, n = pq = 512504565321
    // s in [1, n-1] -> s= 412504565321
    static double xn = (412504565321 * 412504565321) % 512504565321;
    double value = xn;
    xn = fmod((xn * xn), 512504565321); // equivalent to (xn * xn) % M
    return value;
}
```

FIGURE 5.14 – Code C pour le générateur Blum-Blum-Shub

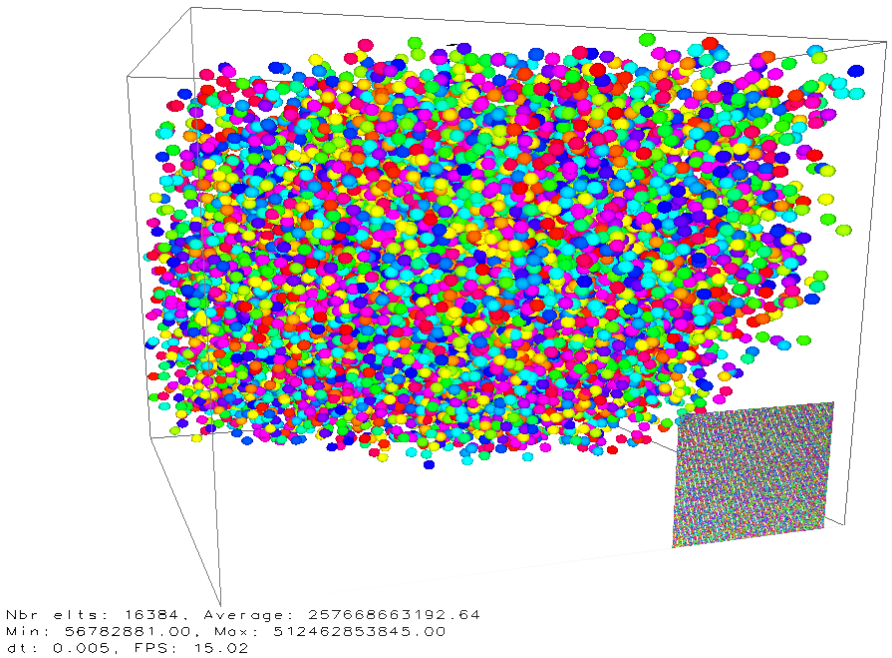


FIGURE 5.15 – Attracteur du générateur Blum-Blum-Shub - vue 45°

Finalement, nous convertissons le résultat en entiers. Ainsi, nous partons d'une séquence linéaire d'entiers pour obtenir une séquence pseudo aléatoire d'entiers. La liste suivante montre le résultat obtenu pour les huit premières entrées de la séquence :

$$
\begin{aligned}
RC4_k(00000200000000000000000000000001) &=_{16} (06080c0e18202929393349576676878 2)\\
&=_{10} (80171508551538131176008731931814480 66)\\
RC4_k(00000200000000000000000000000002) &=_{16} (06080c0e18202929393349576676878 1)\\
&=_{10} (80171508551538131176008731931814480 65)\\
RC4_k(00000200000000000000000000000003) &=_{16} (06080c0e18202929393349576676878 0)\\
&=_{10} (80171508551538131176008731931814480 64)\\
RC4_k(00000200000000000000000000000004) &=_{16} (06080c0e18202929393349576676878 7)\\
&=_{10} (80171508551538131176008731931814480 71)\\
RC4_k(00000200000000000000000000000005) &=_{16} (06080c0e18202929393349576676878 6)\\
&=_{10} (80171508551538131176008731931814480 70)\\
RC4_k(00000200000000000000000000000006) &=_{16} (06080c0e18202929393349576676878 5)\\
&=_{10} (80171508551538131176008731931814480 69)\\
RC4_k(00000200000000000000000000000007) &=_{16} (06080c0e18202929393349576676878 4)\\
&=_{10} (80171508551538131176008731931814480 68)\\
RC4_k(00000200000000000000000000000008) &=_{16} (06080c0e18202929393349576676878b)\\
&=_{10} (80171508551538131176008731931814480 75)
\end{aligned}
$$

Nous avons développé une suite d'outils spécifiques [81, VisCipher3d] pour générer la liste et l'afficher ensuite dans un environnement en trois dimensions à l'instar de la démarche présentée ci-dessus pour les PRBG. Nous obtenons la figure 5.16.

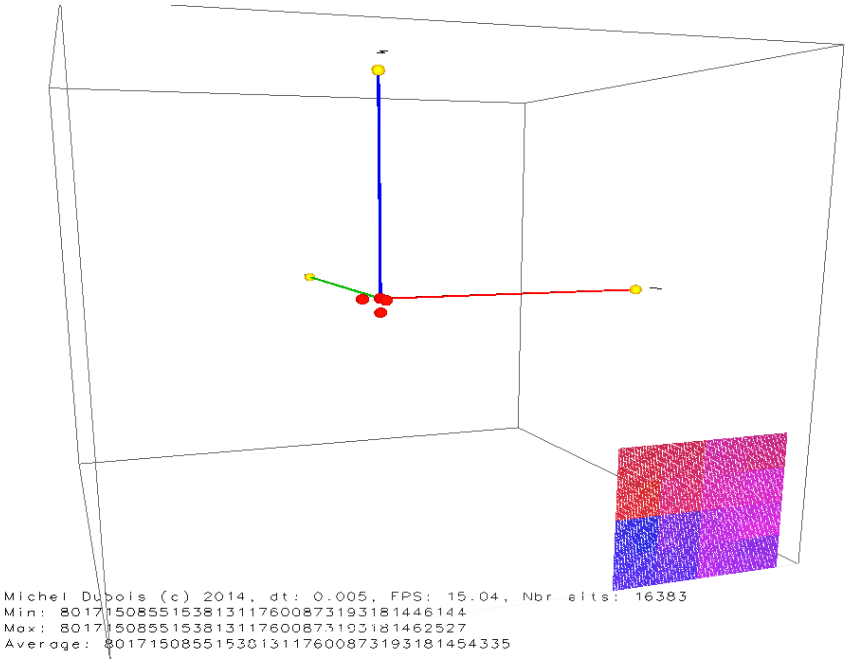


FIGURE 5.16 – Attracteur de RC4 - axe $x \rightarrow x'$

L'attracteur de RC4 est très clairsemé et se résume à quatre boules rouges. La couleur est importante, c'est la couleur de la fin du panel de couleurs. Nous

pouvons en déduire que tous les points de l'attracteur ont les mêmes coordonnées. Nous sommes loin d'une distribution aléatoire dans l'espace comme pour le générateur Blum-Blum-Shub.
Si nous changeons la clef de chiffrement, nous obtenons différents types d'attracteurs : figure 5.17 et figure 5.18. Nous avons donc un lien étroit entre l'attracteur obtenu et la clef de chiffrement utilisée. La représentation de RC4 à l'aide de la courbe de Hilbert montre le même lien entre l'attracteur et la clef utilisée. Plus important encore, la courbe de Hilbert montre que cet algorithme semble loin de l'aléa parfait.

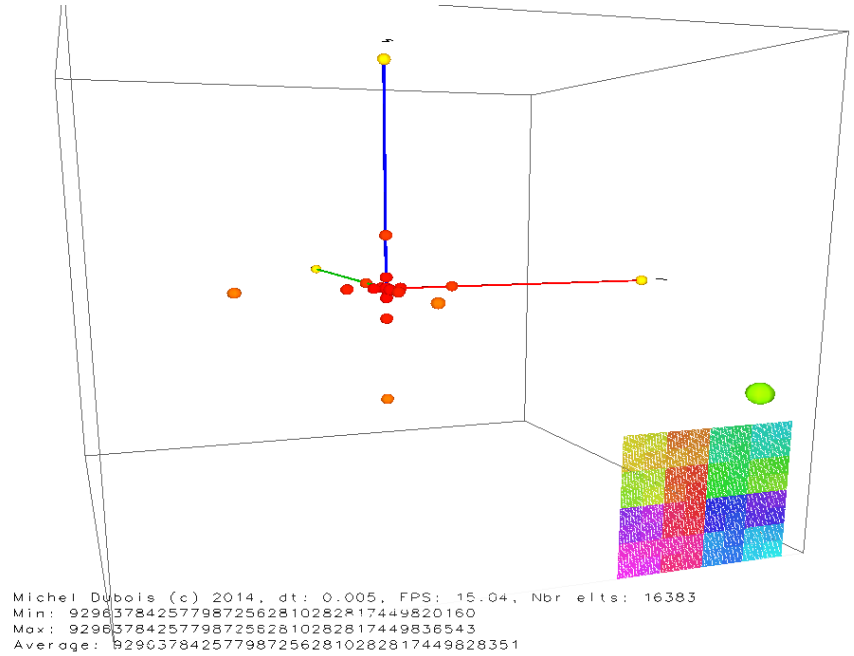


FIGURE 5.17 – Attracteur de RC4 - $k = 5555555555555555$ - axe $x \rightarrow x'$

5.1.4.2 Attracteur de la machine Enigma

La machine Enigma a été inventée par l'ingénieur allemand Arthur Scherbius en 1927 [117]. Le modèle A a été vite abandonné au profit du modèle B, de la taille d'une machine à écrire, puis par une version portable équipée d'indicateurs à lampes avec le modèle C. La machine Enigma et l'entreprise de Scherbius, fondée pour sa commercialisation, vont végéter durant l'entre-deux guerre. La Wehrmacht en achète quelques exemplaires pour évaluation. Ce n'est que lorsque Hitler commence à réarmer l'Allemagne que les experts en cryptologie de la Wehrmacht décident de l'adopter et d'en équiper l'armée allemande.
La machine Enigma est une machine de chiffrement polyalphabétique. Sa première version dispose de 3 rotors comportant chacun 26 positions [142]. Le rotor R_1 tourne chaque fois qu'une touche est pressée, le rotor R_2 tourne selon le mouvement du rotor R_3 qui agit comme un odomètre. Le mouvement des rotors permet de générer une combinaison différente à chaque pression sur une touche du clavier. La clef de chiffrement est définie par la position initiale des 3

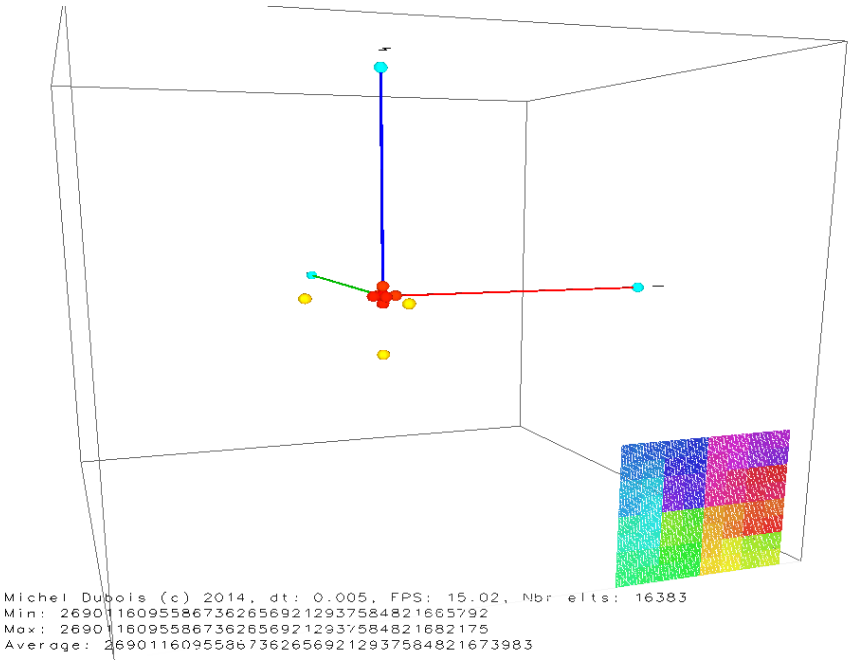


FIGURE 5.18 – Attracteur de RC4 - $k =$ `0f0f0f0f0f0f0f0f` - axe $x \rightarrow x'$

rotors.
Pour calculer l'attracteur de la machine Enigma, nous utilisons une séquence comprenant toutes les combinaisons possibles de 5 caractères de `AAAAA` à `EZZZZ`. Soit $5 \times 26^4 = 2284880$ entrées. Chacune d'entre elles est ensuite chiffrée à l'aide d'un programme reproduisant le fonctionnement d'une machine Engima à trois rotors. La clef utilisée est `AAA`. Le résultat obtenu est ensuite converti en hexadécimal. Ainsi la séquence `AAAAA` devient `FTZMG` $\Rightarrow$ `(0x46545a4d47)` et `EZZZZ` devient `DXILI` $\Rightarrow$ `(0x4458494c49)`.
L'attracteur de la machine Enigma dans un environnement en trois dimensions est présenté dans la figure 5.19 page 106.
L'analyse de l'attracteur de l'Enigma montre un nuage de points aléatoires. Cependant, nous distinguons des alignements et un nuage de points assez épars. Étant donné qu'il y a plus de deux millions de points affichés nous devrions avoir un nuage plus dense.
En zoomant dans le graphique (figure 5.20 page 106), nous voyons que les points sont regroupés en grappes et que leurs couleurs sont similaires. En comptant les points à l'intérieur des grappes, nous trouvons qu'ils contiennent environ 26 points, soit un point pour chaque caractère de l'alphabet. Nous n'avons aucune explication pour ce fait mais une investigation plus approfondie pourrait être intéressante.

5.1.4.3 Attracteur de l'AES

Nous terminons notre comparaison avec l'AES. Pour calculer l'attracteur de l'AES, nous partons de la séquence d'entiers $n = 2^{105}+i$ avec $i = [1 \cdots 1\,000\,000]$ que nous convertissons en blocs de 128 bits. Nous chiffrons ensuite chacun de ces

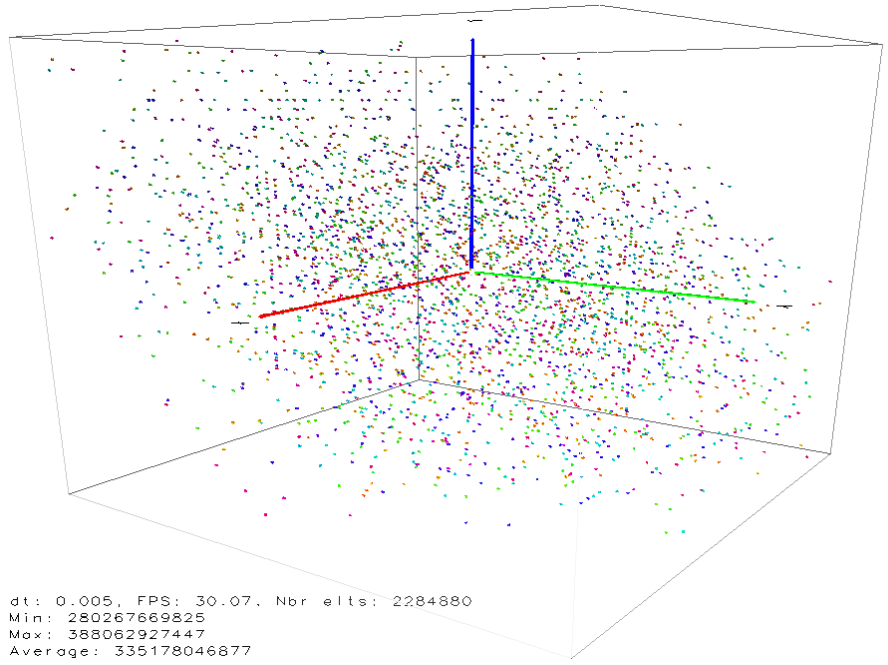


FIGURE 5.19 – Attracteur de la machine Enigma - vue 45°

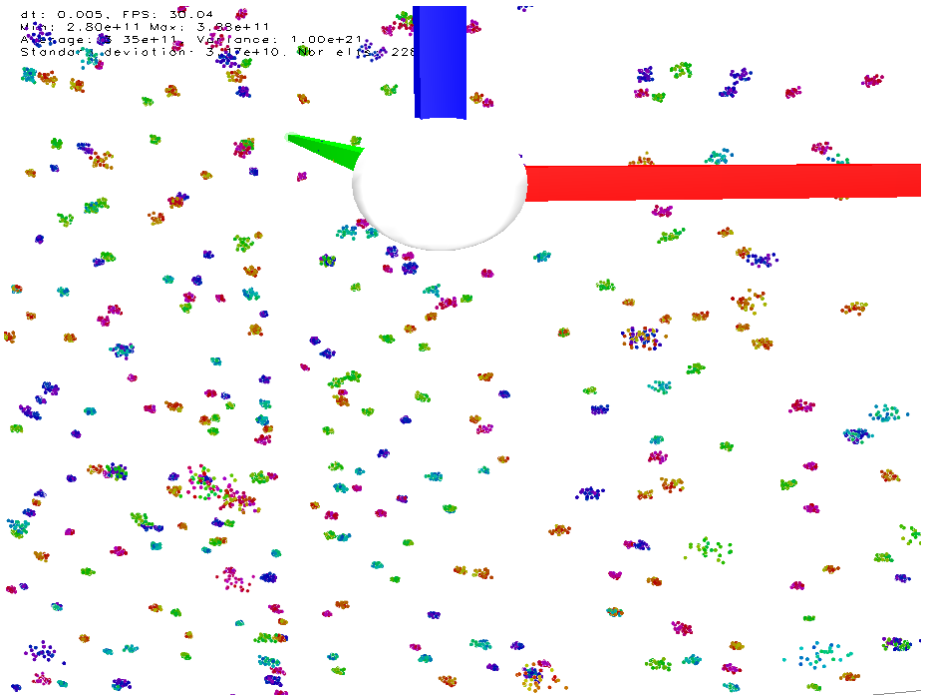


FIGURE 5.20 – Enigma machine attractor - zoomed view

blocs avec la même clef de chiffrement : $k =$ `80000000000000000000000000000000`
Finalement, nous convertissons le résultat obtenu en nombres entiers. Nous partons donc d'une séquence linéaire d'entiers pour obtenir une séquence pseudo-aléatoire d'entiers. La liste de la figure 5.21 page 107 montre le résultat obtenu pour les 16 premières entrées de la séquence.

$$
\begin{aligned}
AES_k(00000200000000000000000000000001) &= (\text{d557cc9f49fe887227b4b9ddece84715})_{16} \\
&= (283581443160616712863761136161128990485)_{10} \\
AES_k(00000200000000000000000000000002) &= (\text{09c7c5a6ebf426a5f19e03bd9040e79d})_{16} \\
&= (13000327896503708020216929648547784605)_{10} \\
AES_k(00000200000000000000000000000003) &= (\text{5f8fc8bf51275ac26522097cb3035f0b})_{16} \\
&= (127023229689953121336747928769328799499)_{10} \\
AES_k(00000200000000000000000000000004) &= (\text{c966ecf33d34f92dc353498769996462})_{16} \\
&= (267709247352391222087383818398701544546)_{10} \\
AES_k(00000200000000000000000000000005) &= (\text{09d800a81f3caa9871aa6ee1ee32ed4e})_{16} \\
&= (130846014035064444883418091784767358 22)_{10} \\
AES_k(00000200000000000000000000000006) &= (\text{723e5071689e7a932405c6d351a87eed})_{16} \\
&= (151855545502638255213403946355100516077)_{10} \\
AES_k(00000200000000000000000000000007) &= (\text{56cfe7cf8cc63cbc4a527ae09957a949})_{16} \\
&= (115393114767635113880023375849033541961)_{10} \\
AES_k(00000200000000000000000000000008) &= (\text{f7ca0f9ad9dfc972e5890cab472d3476})_{16} \\
&= (329368475429008126664105441219366958198)_{10} \\
AES_k(00000200000000000000000000000009) &= (\text{719e9057c4acf3249ee8d48cbeefabb1})_{16} \\
&= (151026074048045206361522544693195287473)_{10} \\
AES_k(0000020000000000000000000000000a) &= (\text{ff6e305fdcebebc447f0dc37c25a86e0})_{16} \\
&= (339525272730300705870076126914789672672)_{10} \\
AES_k(0000020000000000000000000000000b) &= (\text{834edd9eddbf15c1a77c1a99e909ede3})_{16} \\
&= (174538361601988974922403086931709259235)_{10} \\
AES_k(0000020000000000000000000000000c) &= (\text{5344aa331536075f86295762f218bff4})_{16} \\
&= (110682451893361796760390205723301691380)_{10} \\
AES_k(0000020000000000000000000000000d) &= (\text{1b7e76767b1447b707228becfdf25754})_{16} \\
&= (365457880017156978186058074066519550 28)_{10} \\
AES_k(0000020000000000000000000000000e) &= (\text{6907210e0726b85b1c81fe6c0335577b})_{16} \\
&= (139605956066350310999159704129570690939)_{10} \\
AES_k(0000020000000000000000000000000f) &= (\text{e62909d098b6985395aa59b0ae3635a9})_{16} \\
&= (305935522270137279866245407329373009321)_{10}
\end{aligned}
$$

FIGURE 5.21 – Premières entrées de la séquence pour l'AES

En utilisant les mêmes outils que pour l'algorithme de RC4, nous obtenons les figures 5.22 page 108 et 5.23 page 108.
À la différence du générateur Blum-Blum-Shub nous constatons que l'attracteur de l'AES prend la forme d'un nuage uniformément réparti sur les trois axes sous la forme de cubes. Il diffère, en cela, de l'attracteur de l'aléa véritable. Néanmoins, nous pouvons raisonnablement dire que l'AES est un algorithme de chiffrement conçu comme un générateur de bits pseudo-aléatoire, proche de l'aléa véritable.
Après avoir analysé graphiquement le comportement de l'AES, nous allons chercher à voir comment l'algorithme se comporte avec des clefs de chiffrement différentes. Pour cela, nous avons reproduit le processus décrit ci-dessus en utilisant la même séquence initiale et en la chiffrant avec les deux clefs :

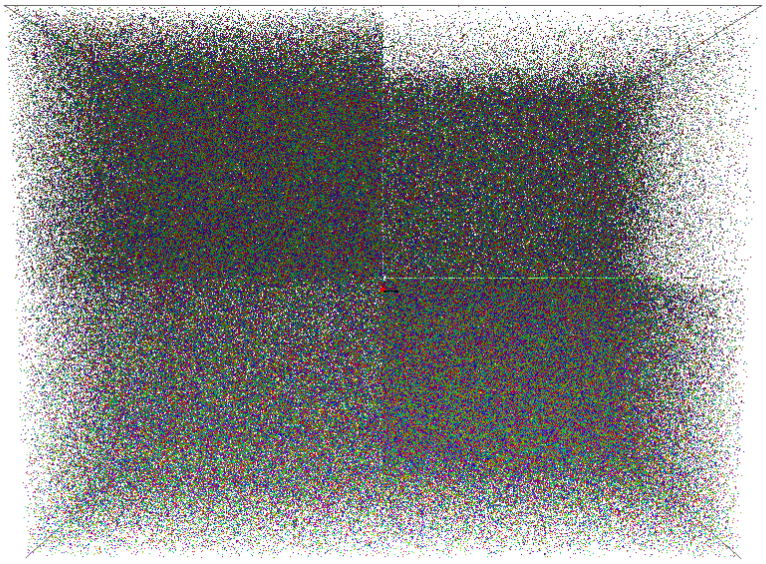


FIGURE 5.22 – Attracteur de l'AES - axe $x \rightarrow x'$

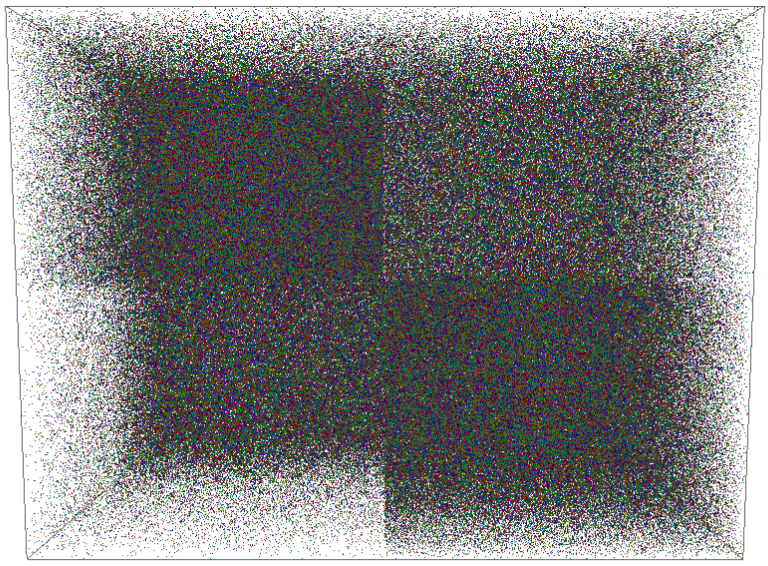


FIGURE 5.23 – Attracteur de l'AES - vue 45°

$$k_1 = \texttt{80000000000000000000000000000000} \tag{5.2}$$

$$k_2 = \texttt{40000000000000000000000000000000} \tag{5.3}$$

Ces deux clefs diffèrent de seulement 1 bit sur les 128 bits les composant.
En affectant ensuite une couleur différente à chaque attracteur, nous pouvons les comparer deux à deux. Le résultat obtenu est présenté dans les figures 5.24 page 109 et 5.25 page 110. Pour générer ces figures nous avons calculé le chiffrement d'une séquence de 450000 blocs identiques.

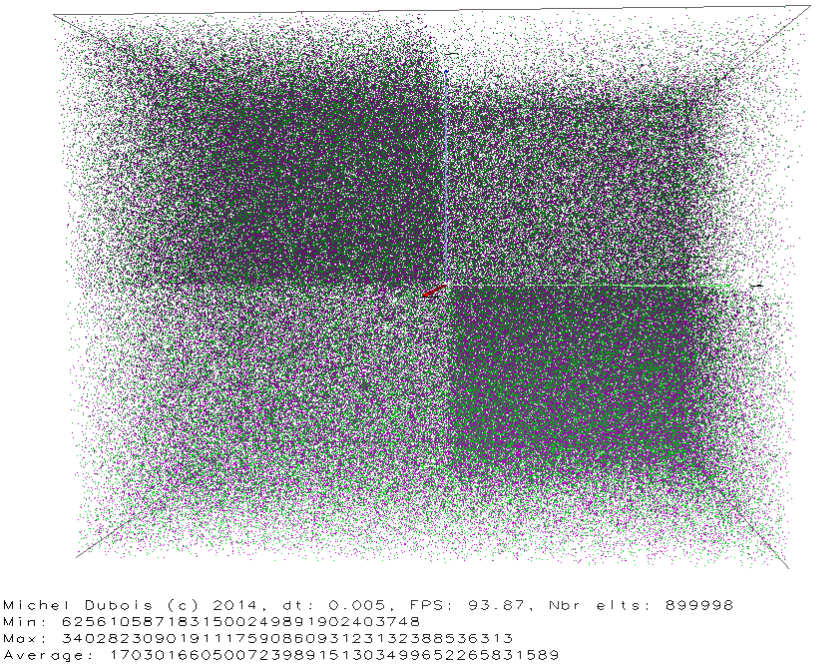


FIGURE 5.24 – Attracteurs de deux chiffrements par l'AES avec des clefs différentes - axe $z \rightarrow z'$

En effectuant la comparaison de ces deux attracteurs, nous constatons qu'ils s'imbriquent étroitement en couvrant uniformément les trois dimensions, toujours en gardant cette répartition en cubes. Cependant, il n'est pas possible de distinguer une possible corrélation entre ces deux ensembles d'éléments. Afin de mieux visualiser une éventuelle corrélation entre les deux ensembles de blocs chiffrés, nous allons représenter le même graphe mais en joignant deux à deux les points correspondants à la même entrée. Ainsi, le point représentant le bloc chiffré avec la clef k_1 du premier bloc de la séquence sera relié au point correspondant au bloc chiffré avec la clef k_2 du même bloc en clair.
Le résultat obtenu est présenté dans les figures 5.26 page 110 et 5.27 page 111. Afin que les graphes soient lisibles, nous avons réduit la taille de l'échantillon à 250 entrées. Le tracé des droites reliant les points est désordonné et ne révèle aucun schéma identifiable.

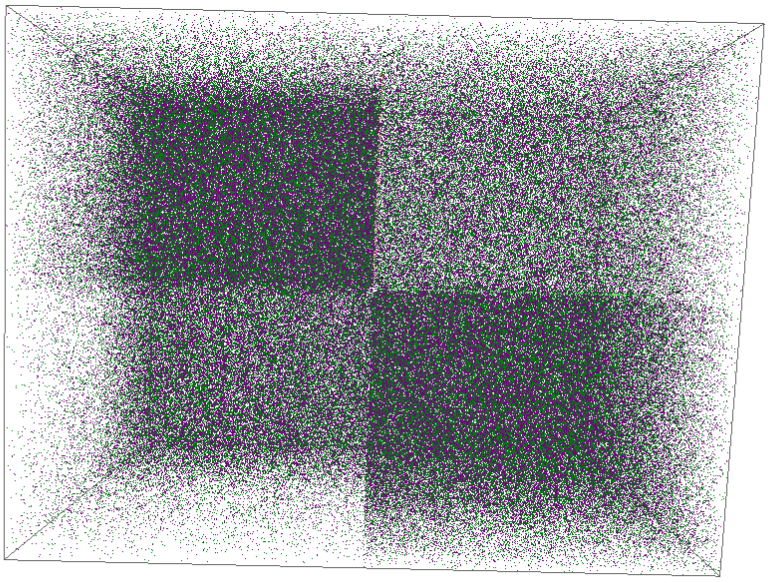


FIGURE 5.25 – Attracteurs de deux chiffrements par l'AES avec des clefs différentes - vue 45°

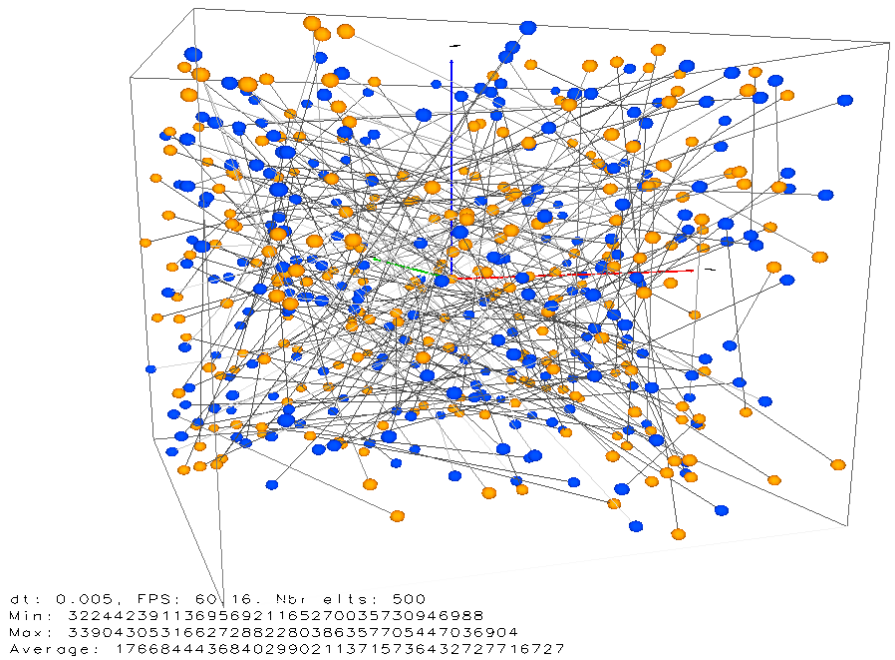


FIGURE 5.26 – Corrélation entre deux attracteurs de l'AES - vue 45°

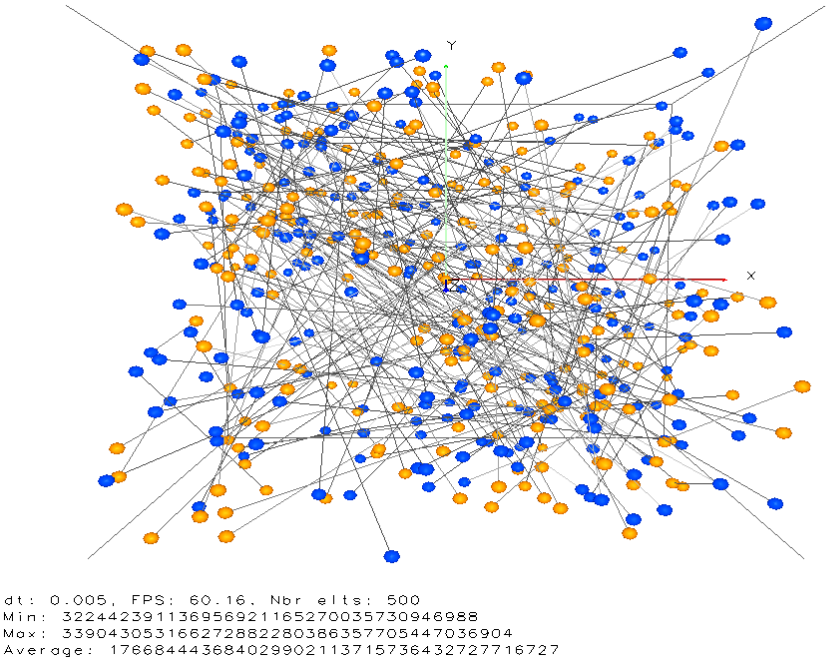


FIGURE 5.27 – Corrélation entre deux attracteurs de l'AES - axe $z \rightarrow z'$

Nous pouvons en déduire raisonnablement qu'il n'y a pas de corrélation entre le chiffrement d'un même bloc de clair avec des clefs de chiffrement différentes. Le calcul des distances de Minkowski [209], avec $p = 2$, pour chaque point du graphe confirme ce résultat. En effet, la liste des distances, dont les premières entrées sont présentées ci-dessous, se comportent comme une suite aléatoire. L'attracteur résultant de la représentation de ces distances en trois dimensions (voir figure 5.28 page 112) a la forme d'un nuage dont la forme spécifique est proche de celle d'un losange. Il serait intéressant de chercher à comprendre ce qui justifie cette forme, mais ce n'est pas l'objet de ce travail.

```
12026847709294981120, 12026869209288865792, 19971070890283577344,
26113153353646714880, 25972774069680922624, 14954846745435789312,
7391670208430327808, 16736310912763425 28, 2444035869612603904,
2802878496097698304, 24268436007518049 28, 17531236510957637 12,
9810861268161230848, 10326031878050832384, 16430236778313869312,
14585735298828013568, 14873569053264308224, 13174990125624895488
```

—=oOo=—

Notre analyse de l'aléa produit par les PRBG et par les algorithmes cryptographiques, bien que visuellement parlante, n'est pas suffisante pour prouver que les séquences produites par ces algorithmes sont statistiquement aléatoires. Cependant, la visualisation de l'aléa, tel que nous l'avons exposé, présente deux

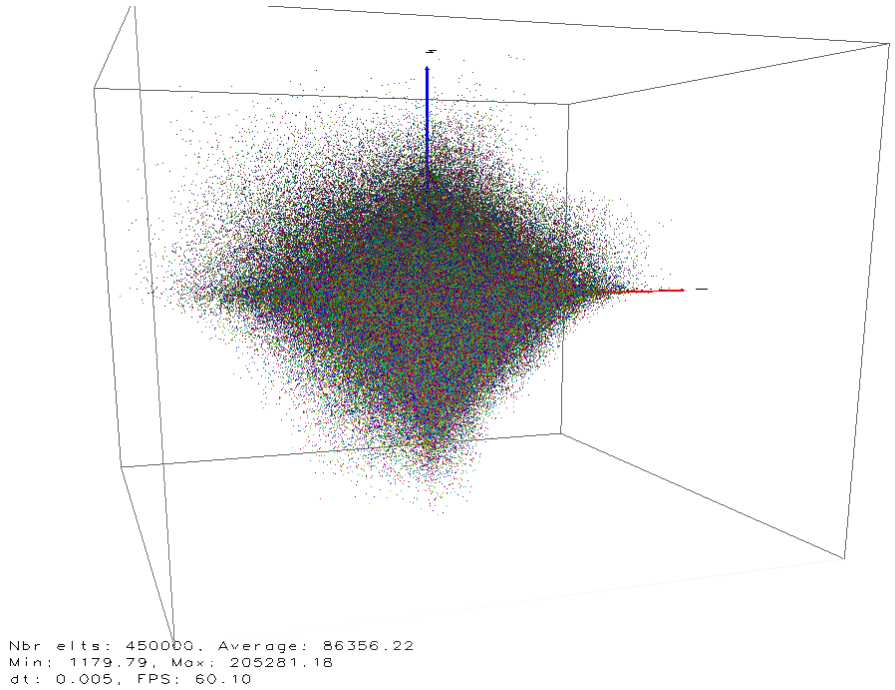


FIGURE 5.28 – Attracteur des distances entre deux processus de chiffrement avec l'AES - vue 45°

avantages intéressants : le premier est qu'elle permet de détecter des patterns et donc ici des problèmes puisque l'on cherche de l'aléatoire, et la deuxième réside dans le fait que notre cerveau ayant plus de facilité pour distinguer rapidement deux images différentes, notre approche nous permet plus facilement de visualiser et différencier un PRBG d'un autre. Ainsi, juste en visualisant dans des environnements en deux et trois dimensions la sortie d'une séquence aléatoire, nous avons la possibilité de détecter d'éventuels patterns et d'identifier le PRBG utilisé pour générer cette séquence. Cette approche nous permet également de réaliser une première analyse rapide d'un potentiel biais sur des algorithmes cryptographiques inconnus.
Ce travail a fait l'objet de publications en France [89] et à l'international [85]. Ce dernier article a reçu le prix du meilleur article et de la meilleure présentation. De nombreux tests statistiques existent pour définir ce qui peut être considéré comme aléatoire. Parmi ces tests, nous pouvons citer le test de fréquence, le test de série, le test d'auto-corrélation, le test statistique universel de Maurer [139],le test de Kolmogorov-Smirnov [135] ou encore la suite de tests du NIST [175]. Aujourd'hui, nous pouvons considérer que les algorithmes de chiffrement symétriques sont statistiquement fiables, dans le sens où ils ont passé avec succès les tests statistiques connus. Le travail du cryptanalyste consiste donc maintenant a trouver un biais statistique exploitable, lié à un défaut de conception inconnu. Dans ce contexte, son objectif est, à partir de nouvelles hypothèses, de développer de nouveaux tests statistiques.

5.2 Test statistique de Möbius

Le test statistique de Möbius est un nouveau test statistique présenté dans [97, 100] et s'appuyant sur le nombre de monômes de degré exactement égal à d dans la forme algébrique normale de toutes les fonctions booléennes modélisant chaque bit produit par un algorithme de chiffrement. Ce test permet de déterminer si ces fonctions booléennes sont réellement aléatoires.

5.2.1 Présentation du test

Le théorème de base sur lequel s'appuie le test statistique de Möbius est le suivant :

Théorème 5.1. Soit la forme algébrique normale d'une fonction booléenne $f : \mathbb{B}_2^n \to \mathbb{B}_2$. Le nombre n de monômes de degré d de f a une distribution normale avec une valeur moyenne E et une variance V donnée par :

$$E_n = \frac{1}{2}C_d^n \quad \text{et} \quad V_n = \frac{1}{4}C_d^n$$

Ainsi, pour des fonctions booléennes de $\mathbb{B}_2^n \to \mathbb{B}_2$ avec $n = (2, 4, 8, 16)$, nous obtenons le tableau de la figure 5.29 page 114.
À titre d'exemple, nous avons généré 16 fonctions booléennes aléatoires de $\mathbb{B}_2^{16} \to \mathbb{B}_2^{16}$. Ensuite, pour chacune de ces fonctions, nous avons calculé leur transformée de Möbius et obtenu ainsi 256 fonctions booléennes de $\mathbb{B}_2^{16} \to \mathbb{B}_2$. Enfin, dans chacune de ces fonctions, nous avons compté le nombre de monômes pour chaque degré $d \in (1, 2, \cdots, 16)$. Le résultat obtenu est formalisé, sous forme de graphe, dans les figures 5.30 page 114 et 5.31 page 115. Nous pouvons constater que ces fonctions suivent bien la répartition décrite dans le théorème 5.1 et formalisée dans le tableau 5.29 page 114.
En partant de l'hypothèse que le nombre de monômes de degré d est distribué selon le théorème 5.1, le test statistique de Möbius affirme que si un algorithme cryptographique passe ce test, alors, il ne présente pas de biais statistique structurel.

5.2.2 Application au mini-AES

Le mini-AES est un algorithme de chiffrement reprenant les mêmes algorithmes que l'AES mais sur 16 bits seulement (Cf. figure 4.10 page 68). Cette taille de bloc et de clef représente un avantage, dans notre cas, puisqu'il est aisé de construire les tables de vérité de l'ensemble d'un tour de chiffrement. En effet, l'espace des clefs et des blocs sont de taille suffisamment réduite pour être traité par un ordinateur conventionnel.

5.2.2.1 Premier tour du mini-AES

Pour rappel, nous avons défini plus haut, que la fonction X_1, décrivant le premier tour du mini-AES, résulte de la conjonction des fonctions `NibbleSub` $NS()$, `ShiftRow` $SR()$ et `MixColumn` $MC()$ de telle façon que nous ayons :

$$X_1(B) = MC \circ SR \circ NS(B)$$

Degré	Taille de bloc							
	2		4		8		16	
	E_2	V_2	E_4	V_4	E_8	V_8	E_{16}	V_{16}
0	0.5	0.25	0.5	0.25	0.5	0.25	0.5	0.25
1	1.0	0.5	2.0	1.0	4.0	2.0	8.0	4.0
2	0.5	0.25	3.0	1.5	14.0	7.0	60.0	30.0
3			2.0	1.0	28.0	14.0	280.0	140.0
4			0.5	0.25	35.0	17.5	910.0	455.0
5					28.0	14.0	2184.0	1092.0
6					14.0	7.0	4004.0	2002.0
7					4.0	2.0	5720.0	2860.0
8					0.5	0.25	6435.0	3217.5
9							5720.0	2860.0
10							4004.0	2002.0
11							2184.0	1092.0
12							910.0	455.0
13							280.0	140.0
14							60.0	30.0
15							8.0	4.0
16							0.5	0.25

FIGURE 5.29 – Répartition du nombre de monômes en fonction de la taille de bloc

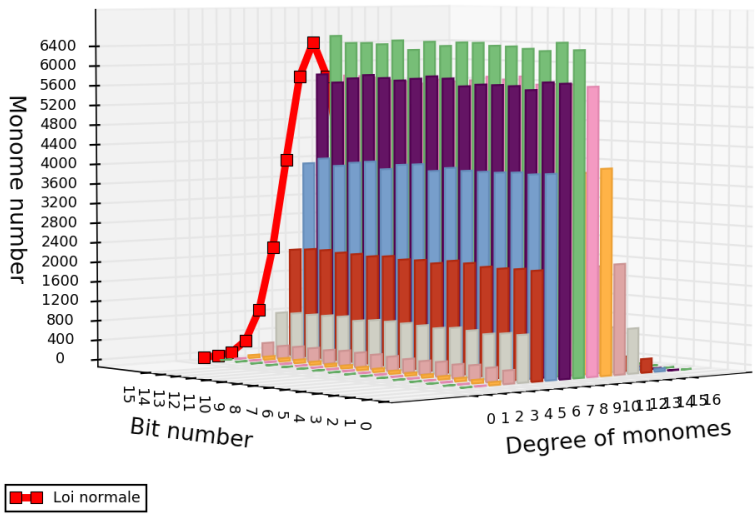


FIGURE 5.30 – Distribution des monômes de degré d (vue 1)

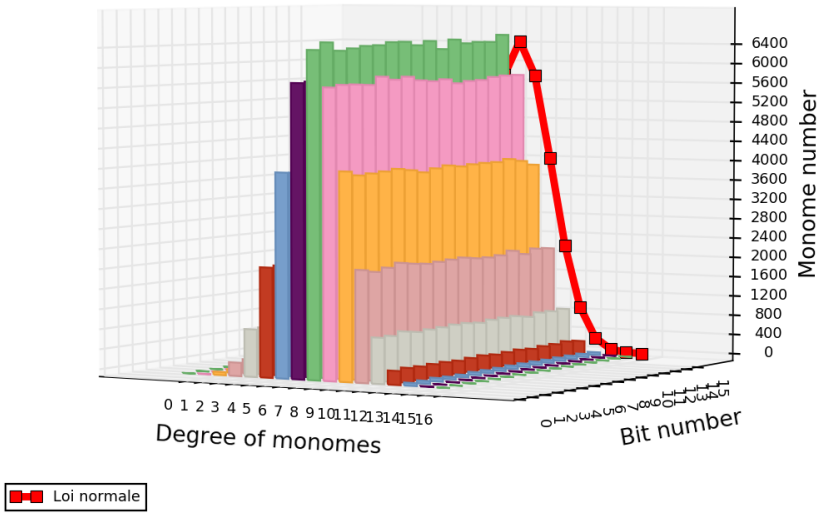


FIGURE 5.31 – Distribution des monômes de degré d (vue 2)

Après avoir généré la table de vérité des fonctions booléennes décrivant chacun des bits de sortie de X_1 et calculé les transformées de Möbius correspondantes, nous obtenons les 16 formes algébriques normales décrivant X_1.
À partir de là, nous pouvons établir le tableau 5.32 page 116 décrivant la répartition du nombre de monômes en fonction du degré d. Le degré maximal des monômes est $d = 3$.
La représentation graphique de la distribution des degrés des monômes de X_1 est donnée dans les figures 5.33 page 116 et 5.34 page 117.
L'analyse du tableau et des figures montrent que le premier round du mini-AES ne passe pas le test statistique de Möbius. En effet, la répartition des monômes n'est pas conforme à celle de notre cas de référence (voir fig. 5.30 p. 114) aussi bien en regardant bit par bit mais également en suivant l'évolution par degré de monômes sur les 16 bits. Par ailleurs, sur ce dernier point, nous constatons une évolution périodique du nombre de monôme de même degré sur les 16 bits.

5.2.2.2 Procédure d'expansion de la clef

Comme pour le premier tour du mini-AES appliquons le test statistique de Möbius à sa procédure d'expansion de la clef. Nous avons précédemment défini les trois fonctions K_1, K_2 et K_3 décrivant le processus de dérivation de la clef tel que, à partir du bloc de clef de 16 bits $K = (k_1, \cdots, k_{16})$, nous ayons les clefs utilisées dans les rounds $k_i = (k_{i,1}, \cdots, k_{i,16})$ avec $i \in (1, 2, 3)$. Nous allons nous intéresser aux fonctions booléennes de la fonction K_3.
En appliquant la même démarche que pour le premier tour, nous obtenons 16 équations à partir desquelles nous pouvons établir le tableau 5.35 page 118 décrivant la répartition du nombre de monômes en fonction du degré d. Le degré

Bit	Nombre de monômes de degrés :			
	0 ($E = 0, 5$)	1 ($E = 8, 0$)	2 ($E = 60, 0$)	3 ($E = 280, 0$)
0	1	3	8	4
1	1	6	9	5
2	1	4	6	6
3	0	6	6	5
4	1	3	8	4
5	1	6	9	5
6	1	4	6	6
7	0	6	6	5
8	1	3	8	4
9	1	6	9	5
10	1	4	6	6
11	0	6	6	5
12	1	3	8	4
13	1	6	9	5
14	1	4	6	6
15	0	6	6	5

FIGURE 5.32 – Tableau de distribution des degrés des monômes pour la fonction X_1 du mini-AES

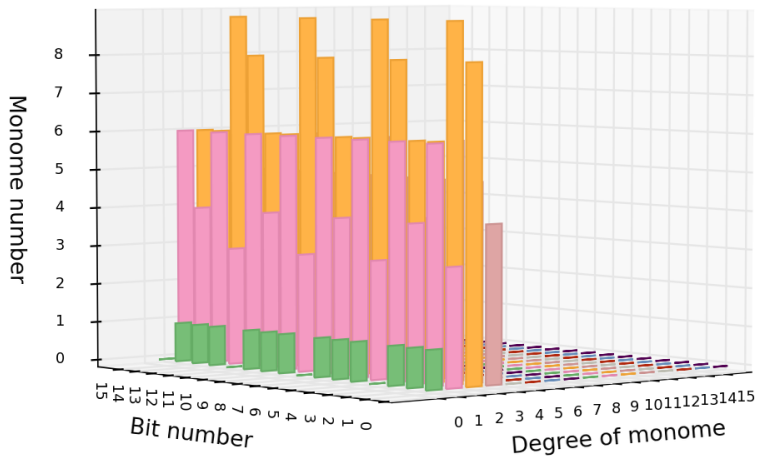


FIGURE 5.33 – Distribution des monômes de degré d pour la fonction X_1 du mini-AES (vue 1)

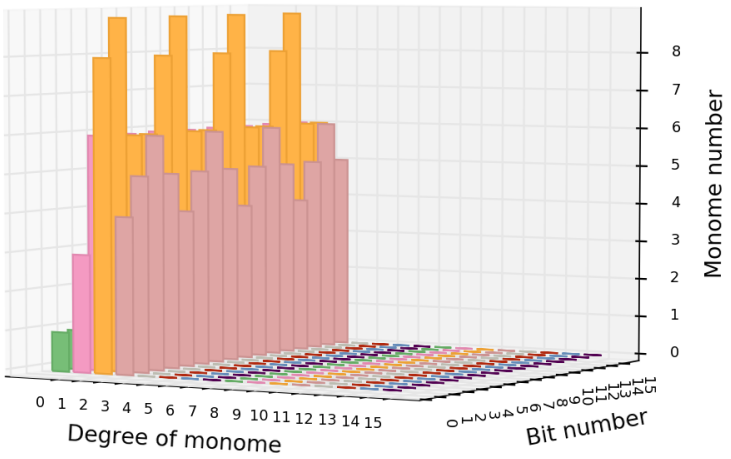


FIGURE 5.34 – Distribution des monômes de degré d pour la fonction X_1 du mini-AES (vue 2)

maximal des monômes est $d = 5$.
La représentation graphique de la distribution des degrés des monômes de K_3 est donnée dans les figures 5.36 page 118 et 5.37 page 119.
Comme pour la fonction X_1, la distribution des degrés des monômes montre de nombreuses disparités par rapport à la distribution normales des monômes de notre cas de référence (voir fig. 5.30 p. 114). De même, comme pour la fonction X_1, nous constatons la présence d'une évolution périodique du nombre de monômes de même degré sur les 16 bits. La période constatée est de 4, ce qui correspond à la taille d'un nibble, brique de base du mini-AES. Nous sommes donc en présence d'un schéma d'évolution plus proche d'une suite périodique que d'un véritable aléa.
En conclusion, nous pourrions penser que le mini-AES ne passe pas le test statistique de Möbius. En effet, chaque fonction interne de l'algorithme, prise séparément, présente des biais statistiques. En fait, faute de puissance de calcul, il est impossible de modéliser l'intégralité du processus de chiffrement du mini-AES. Nous pensons que ces biais statistiques s'estompent par l'imbrication des fonctions internes de l'algorithme et aussi par le jeu des tours. Cependant, cet algorithme ne répond pas complètement au paradigme exposé au début de ce chapitre [1]. En effet, même si le résultat obtenu à l'issue du processus du chiffrement semble proche de l'aléa véritable, il n'en va pas de même des étapes internes.

1. Le Graal de tout algorithme cryptographique est d'obtenir, à chaque étape interne et à l'issue du processus de chiffrement, une séquence semblant la plus proche possible de l'aléa parfait.

Bit	Nombre de monômes de degrés :					
	0 ($E = 0, 5$)	1 ($E = 8$)	2 ($E = 60$)	3 ($E = 280$)	4 ($E = 910$)	5 ($E = 2184$)
0	1	10	62	153	217	87
1	0	13	45	116	103	51
2	1	13	39	163	153	87
3	0	10	52	116	115	51
4	0	9	64	151	217	87
5	1	13	44	117	103	51
6	0	13	39	163	153	87
7	1	12	52	117	115	51
8	1	9	62	153	217	87
9	0	14	45	116	103	51
10	1	12	39	163	153	87
11	0	9	52	116	115	51
12	0	10	64	151	217	87
13	1	12	44	117	103	51
14	0	12	39	163	153	87
15	1	11	52	117	115	51

FIGURE 5.35 – Tableau de distribution des degrés des monômes pour la fonction K_3 du mini-AES

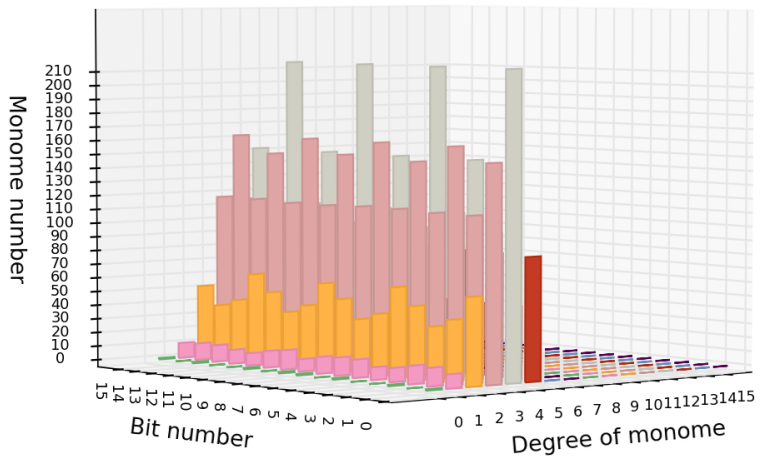


FIGURE 5.36 – Distribution des monômes de degré d pour la fonction K_3 du mini-AES (vue 1)

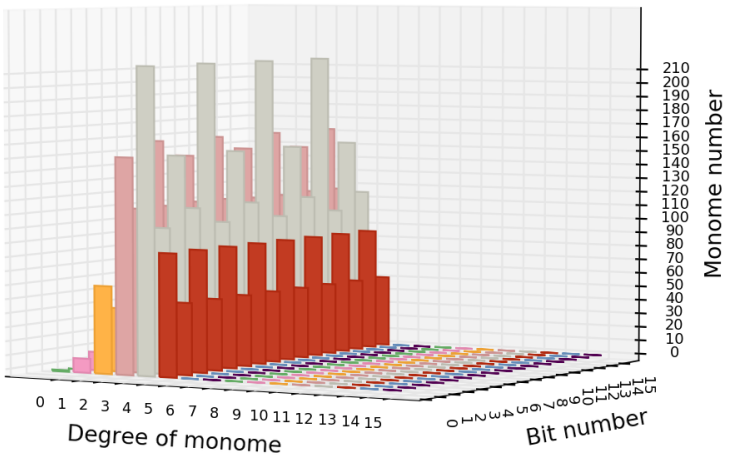


FIGURE 5.37 – Distribution des monômes de degré d pour la fonction K_3 du mini-AES (vue 2)

5.2.3 Application à l'AES

Un algorithme de chiffrement par bloc comme l'AES, utilise une clef K de n bits pour chiffrer un bloc B de m bits. La clef K est elle-même étendue en t clefs de tour de n bits chacune. Cet algorithme peut être représenté par un ensemble de fonctions booléennes tel que le message chiffré C est :

$$C = (c_0, c_1, \cdots, c_{m-1}) = (f_0(K_0, P), f_1(K_1, P), \cdots, f_{n-1}(K_{t-1}, P))$$

Chacune de ces fonctions $(f_0, \cdots, f_{n-1})$ étant des fonctions booléennes $\mathbb{B}_2^{n+m} \rightarrow \mathbb{B}_2$. Dans le cas de l'AES-128 nous avons $n = 128$, $m = 128$ et $t = 10$.
La démarche que nous avons mise en place au début de ce chapitre nous a permis de générer les 128 fonctions booléennes décrivant les bits de sortie de l'*Advanced Encryption Standard*. À partir des fichiers découlant de la forme algébrique normale de ces fonctions, il est maintenant plus aisé d'appliquer le test statistique de Möbius à l'AES.

5.2.3.1 Les fonctions de tour

La fonction `SubBytes` applique une substitution non-linéaire à chaque octet du tableau d'état en utilisant une table de substitution (S-Box).
La représentation de la distribution des degrés des monômes de chacune des 128 formes algébriques normales de la fonction `SubBytes` est détaillée dans les figures 5.38 page 120, 5.39 page 121, 5.40 page 121 et 5.41 page 122. Afin d'obtenir ce résultat, pour chaque équation, nous calculons le nombre de monômes et

leur répartition par degré. Nous obtenons le résultat suivant (présenté sur les 7 derniers bits) :

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	110	$[0, 4, 10, 26, 33, 24, 11, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	112	$[1, 3, 10, 31, 30, 24, 11, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	114	$[1, 3, 7, 32, 32, 27, 10, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	131	$[0, 5, 17, 29, 32, 34, 11, 3, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	136	$[0, 4, 11, 29, 39, 33, 16, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	145	$[0, 4, 12, 38, 36, 33, 17, 5, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	133	$[1, 4, 14, 30, 33, 31, 16, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	132	$[1, 4, 16, 30, 33, 30, 15, 3, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

Nous constatons que sur l'ensemble des monômes, le degré maximal est $d = 7$.

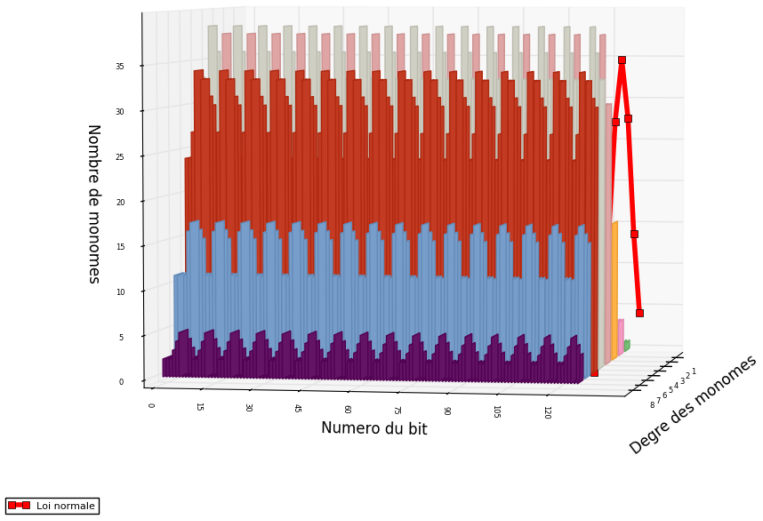


FIGURE 5.38 – Distribution des monômes de degré d pour la fonction SubBytes de l'AES (vue 1)

Afin de déterminer de façon formelle si la fonction `SubBytes` passe le test statistique de Möbius, nous allons tester l'adéquation de la distribution normale des degrés décrite dans le théorème 5.1 et celle observée dans les formes algébriques normales des 128 fonctions booléennes décrivant les bits de sortie de la fonction `subBytes`.
Pour cela, nous calculons, pour chaque degré, la distribution prévue des degrés $d_n = \frac{1}{2}C_d^n$ avec $n = 8$, puis, celle obtenue à partir des 128 formes algébriques normales de la fonction, nous l'appelons $\hat{d}_n$. Ensuite nous calculons la statistique T telle que :

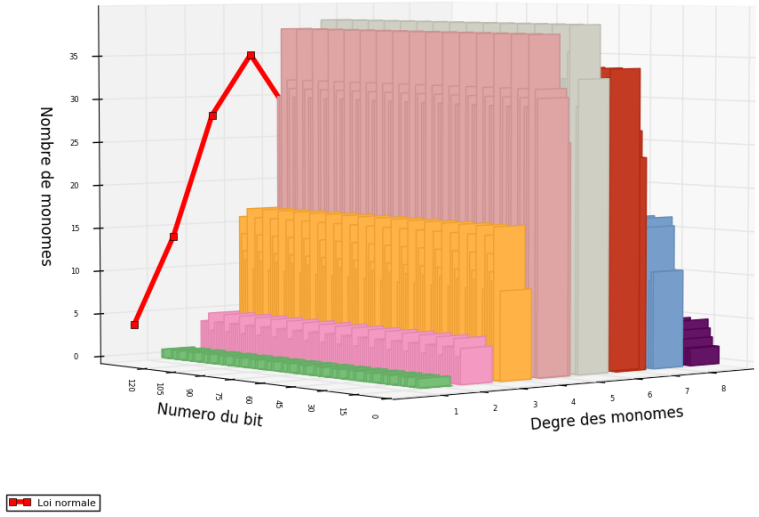


FIGURE 5.39 – Distribution des monômes de degré d pour la fonction SubBytes de l'AES (vue 2)

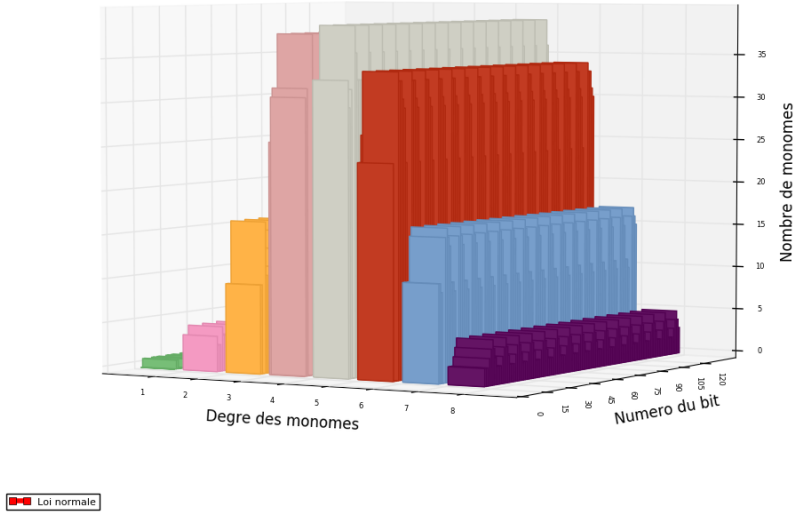


FIGURE 5.40 – Distribution des monômes de degré d pour la fonction SubBytes de l'AES (vue 3)

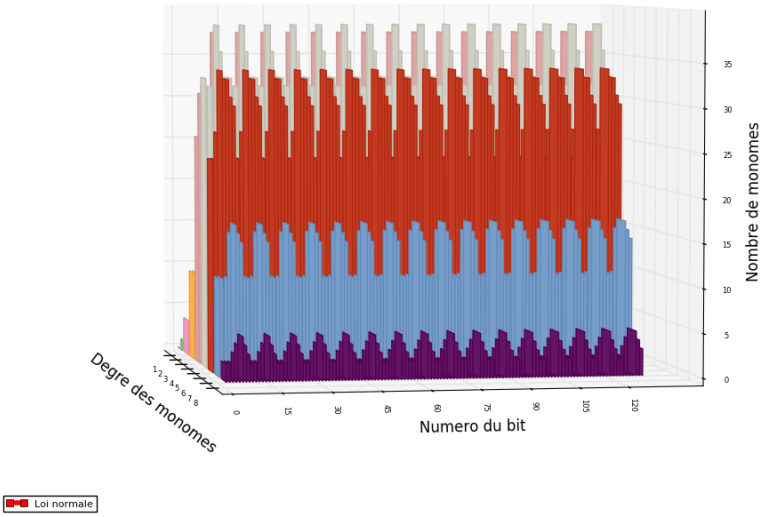


FIGURE 5.41 – Distribution des monômes de degré d pour la fonction SubBytes de l'AES (vue 4)

$$T = \sum_{i=1}^{128} \frac{(\hat{d}_n^i - d_n^i)^2}{d_n^i}$$

Cette statistique suit alors une loi du χ^2 à 7 degrés de liberté dont les valeurs théoriques sont : $\chi^2 = 154,30$ pour $\alpha = 0,050$, $\chi^2 = 166,99$ pour $\alpha = 0,010$, $\chi^2 = 181,99$ pour $\alpha = 0,001$.
Nous obtenons alors le tableau suivant [2] :

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	64.0	passe	passe	passe
$d = 1$	12.0	passe	passe	passe
$d = 2$	122.29	passe	passe	passe
$d = 3$	79.43	passe	passe	passe
$d = 4$	32.91	passe	passe	passe
$d = 5$	75.43	passe	passe	passe
$d = 6$	69.71	passe	passe	passe
$d = 7$	60.0	passe	passe	passe
$d = 8$	64.0	passe	passe	passe

Si nous appliquons le même raisonnement à la fonction `ShiftRows`, après le calcul du nombre de monômes et leur répartition par degré, nous obtenons le résultat suivant (présenté sur les 7 derniers bits) :

2. Le test est validé si la valeur obtenue pour T est inférieure à la valeur de référence de χ^2.

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	110	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	112	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	114	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	131	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	136	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	145	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	133	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	132	$[0, 1, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

En appliquant ensuite le test statistique de Möbius nous obtenons le tableau suivant :

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	64.0	passe	passe	passe
$d = 1$	288.0	échec	échec	échec
$d = 2$	1792.0	échec	échec	échec
$d = 3$	3584.0	échec	échec	échec
$d = 4$	4480.0	échec	échec	échec
$d = 5$	3584.0	échec	échec	échec
$d = 6$	1792.0	échec	échec	échec
$d = 7$	512.0	échec	échec	échec
$d = 8$	64.0	passe	passe	passe

De même, pour la fonction `MixColumns`, après le calcul du nombre de monômes et leur répartition par degré, nous obtenons le résultat suivant (présenté sur les 7 derniers bits) :

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	110	$[0, 5, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	112	$[0, 5, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	114	$[0, 5, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	131	$[0, 7, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	136	$[0, 7, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	145	$[0, 5, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	133	$[0, 7, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	132	$[0, 5, 0, 0, 0, 0, 0, 0, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

En appliquant ensuite le test statistique de Möbius nous obtenons le tableau suivant

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	64.0	passe	passe	passe
$d = 1$	128.0	passe	passe	passe
$d = 2$	1792.0	échec	échec	échec
$d = 3$	3584.0	échec	échec	échec
$d = 4$	4480.0	échec	échec	échec
$d = 5$	3584.0	échec	échec	échec
$d = 6$	1792.0	échec	échec	échec
$d = 7$	512.0	échec	échec	échec
$d = 8$	64.0	passe	passe	passe

En conclusion, nous pouvons dire que la fonction `SubBytes` ne présente pas de biais statistique au regard du test statistique de Möbius. Par contre les fonctions

`ShiftRows` et `MixColumns` semblent en présenter. En réalité, cette différence de résultat s'explique par le fait que dans le tour de l'AES seule la fonction `SubBytes` fait de la permutation sur les bits et assure donc la fonction de diffusion dans l'algorithme. Les deux autres fonctions ne font que de la substitution de bits ce qui explique que le degré maximal des fonctions soit de 1. Pour les fonctions `ShiftRows` et `MixColumns`, il ne s'agit donc pas d'un biais statistique.

5.2.3.2 Tours de chiffrement et de déchiffrement

Nous allons appliquer maintenant la même démarche à un tour de chiffrement de l'AES. À l'instar du mini-AES, nous réduisons le tour de chiffrement à la composition des fonctions `SubBytes`, `ShiftRows` et `MixColumns`. La fonction résultante est donc $T(B) = MC \circ SR \circ NS(B)$.
La représentation de la distribution des degrés des monômes de chacune des 128 formes algébriques normales de la fonction $T(B)$ est détaillée dans l'annexe D page 175 et dans les figures 5.42 page 125, 5.43 page 125, 5.44 page 126 et 5.45 page 126. Comme pour la fonction `SubBytes`, pour chaque équation, nous calculons le nombre de monômes et leur répartition par degré. Nous obtenons le résultat suivant (présenté sur les 7 derniers bits) :

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	554	$[2, 18, 50, 140, 159, 120, 55, 10, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	564	$[5, 15, 44, 157, 154, 126, 53, 10, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	604	$[3, 19, 55, 154, 160, 149, 52, 12, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	885	$[0, 31, 93, 197, 240, 216, 87, 21, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	918	$[0, 28, 77, 215, 255, 213, 104, 26, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	701	$[2, 20, 64, 174, 174, 161, 83, 23, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	883	$[5, 28, 94, 202, 231, 201, 100, 22, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	616	$[3, 20, 68, 142, 165, 138, 67, 13, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

Nous constatons que sur l'ensemble des monômes, le degré maximal est $d = 7$.
En calculant la fonction T décrite plus haut nous obtenons également une statistique qui suit une loi du χ^2 à 7 degrés de liberté dont les valeurs théoriques sont : $\chi^2 = 154,30$ pour $\alpha = 0,05$, $\chi^2 = 166,99$ pour $\alpha = 0,01$, $\chi^2 = 181,99$ pour $\alpha = 0,001$.
Nous pouvons maintenant établir le tableau suivant :

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	1856.0	échec	échec	échec
$d = 1$	11740.0	échec	échec	échec
$d = 2$	29603.43	échec	échec	échec
$d = 3$	98959.43	échec	échec	échec
$d = 4$	96102.4	échec	échec	échec
$d = 5$	92569.14	échec	échec	échec
$d = 6$	37761.14	échec	échec	échec
$d = 7$	6700.0	échec	échec	échec
$d = 8$	64.0	passe	passe	passe

Comme nous l'avons vu précédemment, le tour de déchiffrement insère le `XOR` avec la clef de tour entre les fonctions `InvShiftRows` et `InvMixColumns`. La fonction de tour ne permet donc de combiner que les équations `InvSubBytes` et `InvShiftRows`. La fonction résultante est donc $T(B) = ISB \circ ISR(B)$.

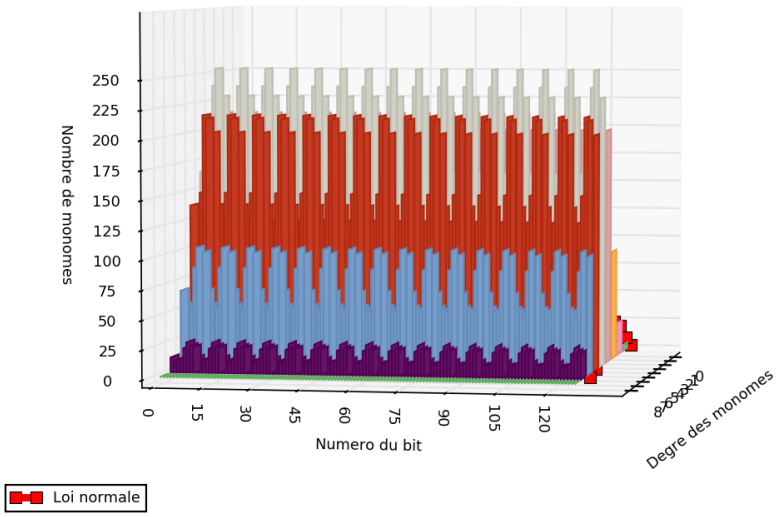


FIGURE 5.42 – Distribution des monômes de degré d pour un tour de chiffrement de l'AES (vue 1)

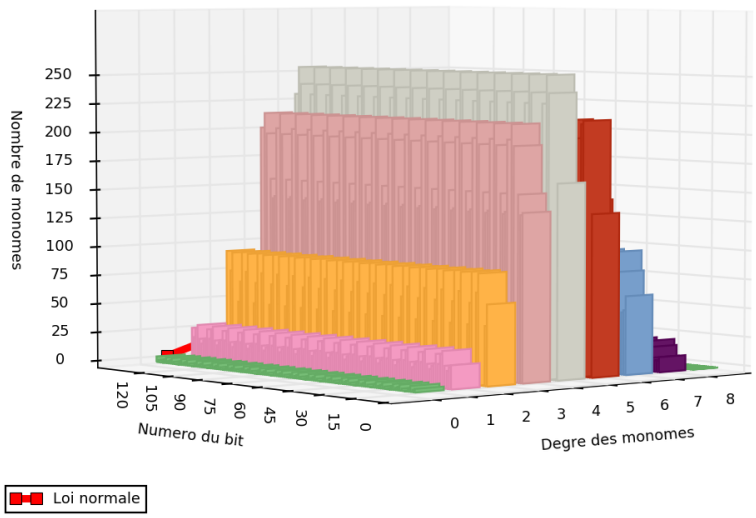


FIGURE 5.43 – Distribution des monômes de degré d pour un tour de chiffrement de l'AES (vue 2)

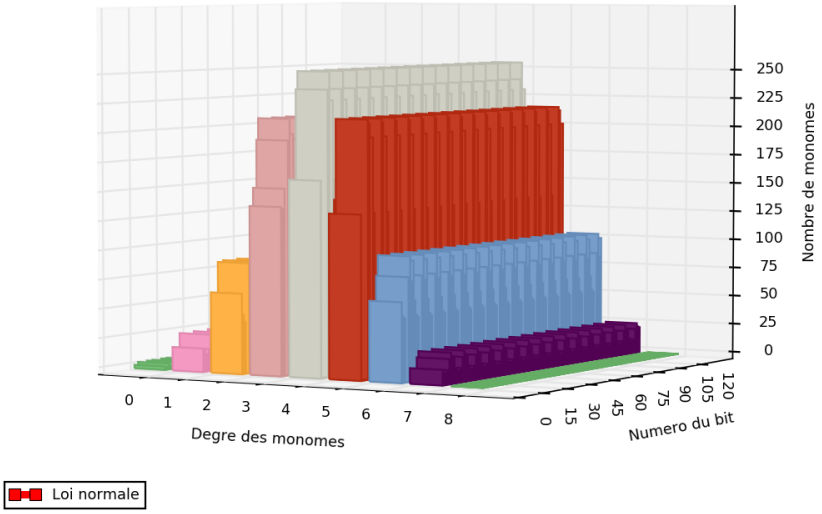


FIGURE 5.44 – Distribution des monômes de degré d pour un tour de chiffrement de l'AES (vue 3)

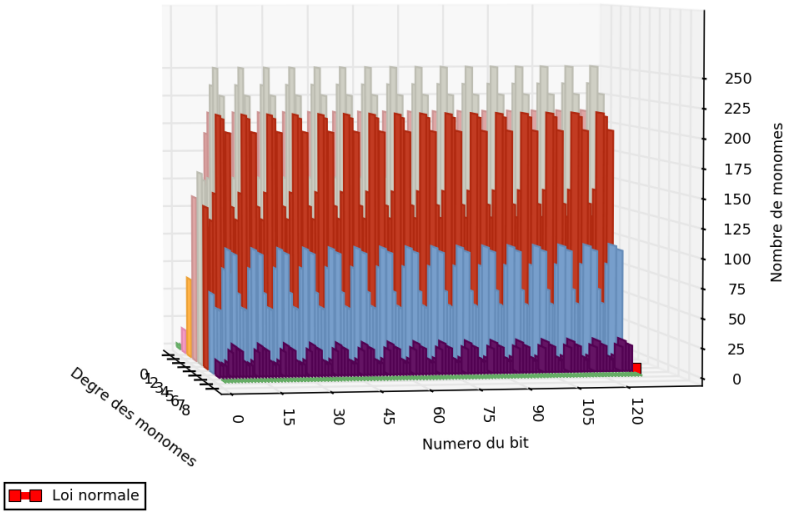


FIGURE 5.45 – Distribution des monômes de degré d pour un tour de chiffrement de l'AES (vue 4)

Appliquons maintenant le raisonnement précédent à cette fonction de tour de déchiffrement. Nous commençons par calculer le nombre de monômes et leur répartition par degré. Nous obtenons le résultat suivant (présenté sur les 7 derniers bits) :

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	110	$[0, 1, 14, 21, 32, 27, 14, 1, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	129	$[1, 4, 15, 30, 31, 30, 17, 1, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	121	$[0, 6, 14, 29, 27, 30, 11, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	137	$[1, 2, 16, 29, 39, 30, 16, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	117	$[0, 5, 8, 25, 35, 31, 9, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	143	$[0, 3, 17, 33, 41, 31, 13, 5, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	132	$[1, 4, 8, 26, 41, 31, 16, 5, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	115	$[0, 2, 12, 25, 31, 29, 15, 1, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

Nous constatons que sur l'ensemble des monômes, le degré maximal est $d = 7$. En appliquant ensuite le test statistique de Möbius nous obtenons le tableau suivant :

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	64.0	passe	passe	passe
$d = 1$	92.0	passe	passe	passe
$d = 2$	102.86	passe	passe	passe
$d = 3$	58.29	passe	passe	passe
$d = 4$	88.23	passe	passe	passe
$d = 5$	23.43	passe	passe	passe
$d = 6$	60.57	passe	passe	passe
$d = 7$	116.0	passe	passe	passe
$d = 8$	64.0	passe	passe	passe

Nous constatons un comportement différent entre les tours de chiffrement et de déchiffrement. Cette différence est due, d'une part au fait que pour le déchiffrement, nous effectuons notre test sur des fonctions booléennes qui ne décrivent qu'une partie du tour et, d'autre part, au fait que le nombre de monômes des fonctions booléennes ainsi que leur degré maximal influent sur le résultat du test statistique de Möbius. À ce point de notre analyse, nous ne pouvons donc pas nous prononcer sur l'existence d'un éventuel biais statistique sur les fonctions de tour.

5.2.3.3 Fonction d'expansion de la clef

Appliquons maintenant notre démarche à la fonction d'expansion de la clef. Les fonctions booléennes que nous avons décrites correspondent à la construction d'une clef de tour. Ces fonctions sont identiques pour le processus de chiffrement et pour celui de déchiffrement.
La représentation de la distribution des degrés des monômes de chacune des 128 formes algébriques normales de la fonction d'expansion d'un tour de clef est détaillée dans les figures 5.46 page 128, 5.47 page 129, 5.48 page 129 et 5.49 page 130.
Pour chaque équation, le calcul du nombre de monômes et leur répartition par degré donne le résultat suivant (présenté sur les 7 derniers bits) :

Bit	Nbr. monômes	Répartition des degrés	Répartition attendue
120	114	$[0, 8, 10, 26, 33, 24, 11, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
121	116	$[1, 7, 10, 31, 30, 24, 11, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
122	118	$[1, 7, 7, 32, 32, 27, 10, 2, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
123	135	$[0, 9, 17, 29, 32, 34, 11, 3, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
124	140	$[0, 8, 11, 29, 39, 33, 16, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
125	149	$[0, 8, 12, 38, 36, 33, 17, 5, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
126	137	$[1, 8, 14, 30, 33, 31, 16, 4, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$
127	136	$[1, 8, 16, 30, 33, 30, 15, 3, 0]$	$[0.5, 4, 14, 28, 35, 28, 14, 4, 0.5]$

Nous constatons que sur l'ensemble des monômes, le degré maximal est $d = 7$.

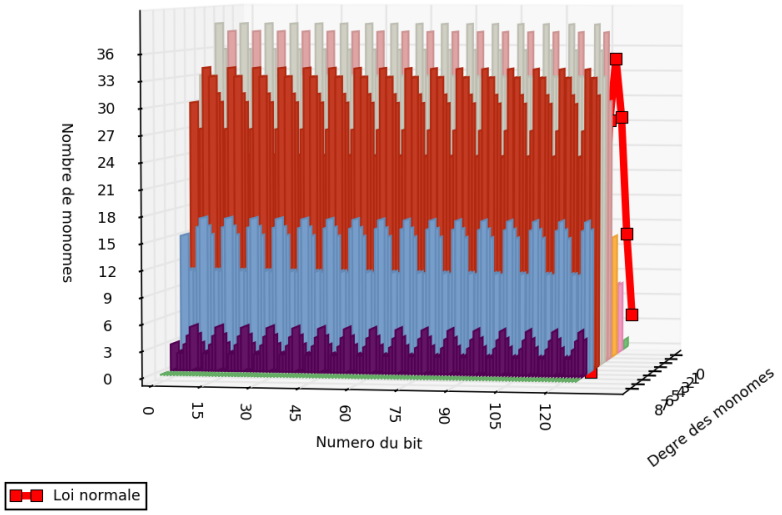


FIGURE 5.46 – Distribution des monômes de degré d pour la fonction d'expansion de la clef de l'AES (vue 1)

Enfin, en calculant la fonction T décrite plus haut nous obtenons également une statistique qui suit une loi du χ^2 à 7 degrés de liberté dont les valeurs théoriques sont : $\chi^2 = 154,30$ pour $\alpha = 0,05$, $\chi^2 = 166,99$ pour $\alpha = 0,01$, $\chi^2 = 181,99$ pour $\alpha = 0,001$.
Nous pouvons maintenant établir le tableau suivant :

Degré	T_i	$\alpha = 0,05$	$\alpha = 0,01$	$\alpha = 0,001$
$d = 0$	64.0	passe	passe	passe
$d = 1$	232.0	échec	échec	échec
$d = 2$	122.29	passe	passe	passe
$d = 3$	79.43	passe	passe	passe
$d = 4$	32.91	passe	passe	passe
$d = 5$	75.43	passe	passe	passe
$d = 6$	69.71	passe	passe	passe
$d = 7$	60.0	passe	passe	passe
$d = 8$	64.0	passe	passe	passe

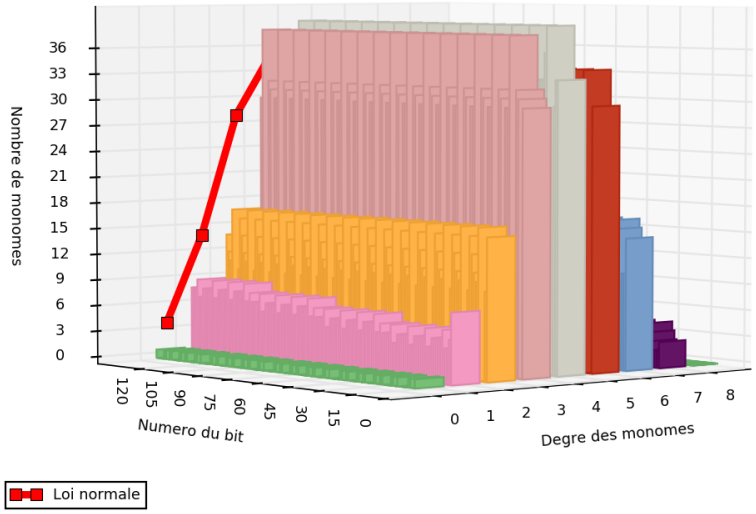


FIGURE 5.47 – Distribution des monômes de degré d pour la fonction d'expansion de la clef de l'AES (vue 2)

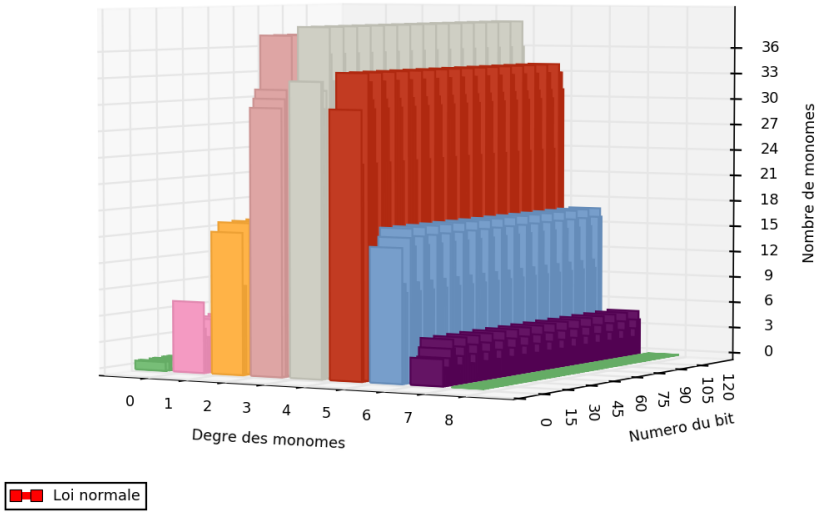


FIGURE 5.48 – Distribution des monômes de degré d pour la fonction d'expansion de la clef de l'AES (vue 3)

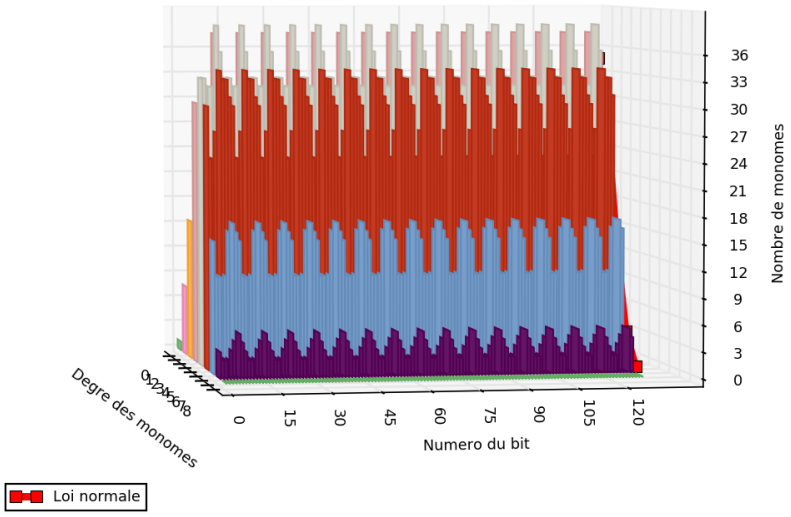


FIGURE 5.49 – Distribution des monômes de degré d pour la fonction d'expansion de la clef de l'AES (vue 4)

En conclusion, nous pouvons dire que la distribution des degrés des monômes de la fonction d'expansion de la clef de tour de l'AES ne présente pas de biais statistique sauf pour le cas ou $d = 1$. Dans ce dernier cas, il existe un biais statistique au regard du test statistique de Möbius. Il peut s'expliquer par le fait qu'à chaque étape de la fonction, l'algorithme utilise un bloc du tour précédent. En effet, la fonction d'expansion de la clef travaille sur un bloc de 4 octets en entrée et fournit un bloc de 4 octets en sortie, hors le bloc obtenu en sortie ne dépend pas uniquement du bloc d'entrée mais également du bloc du tour précédent avec lequel il est `XOR`é. Cette opération est la source du surnombre de monômes de degré 1.

5.3 Analyse structurelle des équations booléennes de l'AES

La combinatoire, est la branche des mathématiques qui étudie le dénombrement, la combinaison et la permutation d'ensembles d'éléments et les relations mathématiques qui caractérisent leurs propriétés.

5.3.1 Dénombrement des monômes

Au travers du test de Möbius nous avons mis en évidence un certain nombre de biais statistiques compensés par le jeu des tours. Attachons nous maintenant à compter et combiner les monômes des équations booléennes que nous avons

obtenues afin d'essayer de confirmer ou d'infirmer ces biais.
Pour cela, commençons par compter le nombre de monômes pour chacun des processus de chiffrement et de déchiffrement. Le graphe obtenu est présenté dans la figure 5.50 page 131. Ce graphe détaille le nombre de monômes de chacune des équations décrivant le processus de chiffrement et de déchiffrement de l'AES. Le nombre moyen de monômes est de 7881 pour le chiffrement et de 2711 pour le déchiffrement.

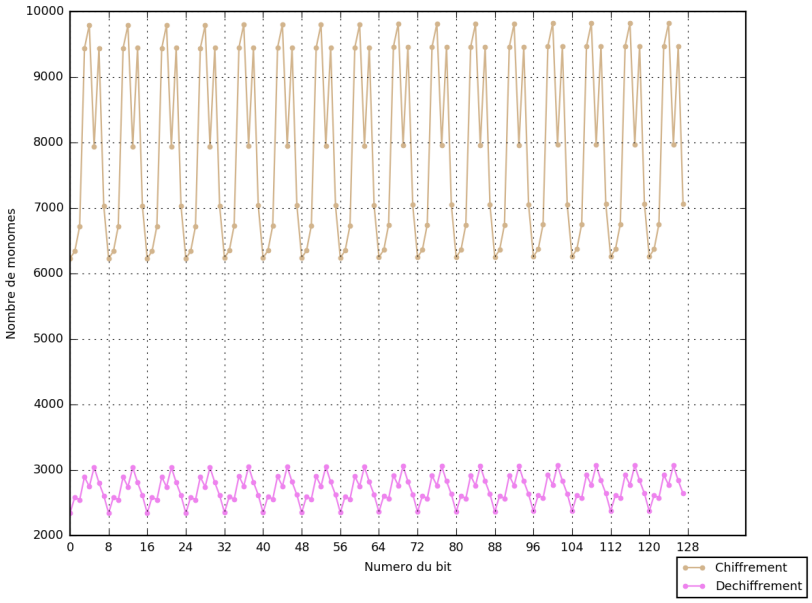
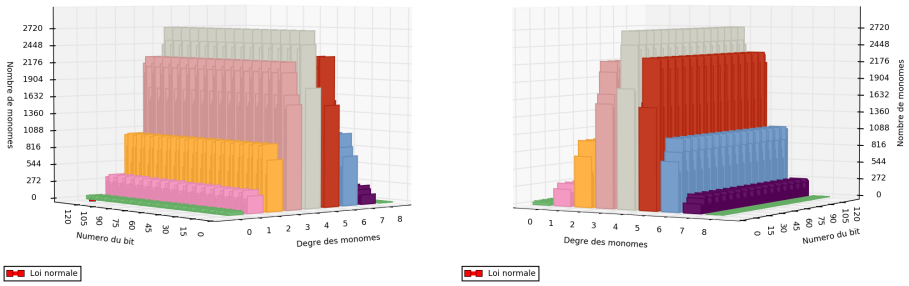


FIGURE 5.50 – Distribution des monômes

Ce graphe met en évidence trois éléments marquants.
Tout d'abord, nous constatons, sur les deux courbes, une évolution cyclique du nombre de monômes. Ce cycle a une période de 8, ce qui correspond au fait que l'élément de base de l'AES est l'octet. Ce premier point est donc normal.
Le deuxième constat, est que l'amplitude des courbes est plus importante pour le chiffrement que pour le déchiffrement. Ce point révèle un nombre de monômes réparti de façon plus resserrée pour le processus de déchiffrement que pour celui de chiffrement. Enfin, le troisième fait notable est la différence importante entre le nombre de monômes des processus de chiffrement et de déchiffrement.
Le processus de déchiffrement étant la fonction inverse de la fonction de chiffrement il vient naturellement à l'esprit que les fonctions décrivant ces deux processus sont similaires et que le nombre et la répartition des monômes de leurs fonctions respectives soient équivalentes. Nous constatons ici qu'il n'en est rien. Sachant que ce sont les équations du processus de déchiffrement qui sont les plus intéressantes pour la cryptanalyse d'un message chiffré cette différence n'est pas anodine.

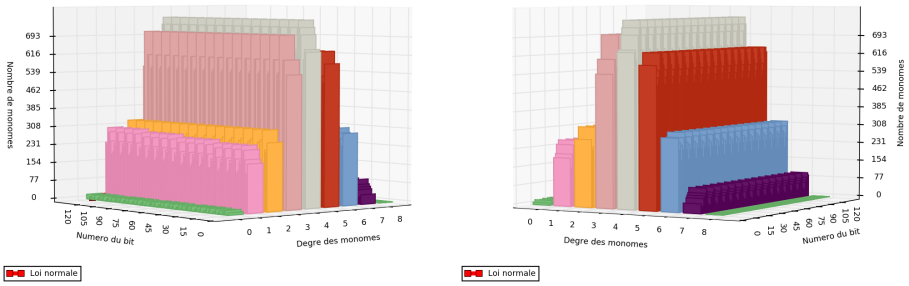
5.3.2 Dénombrement des degrés des monômes

Comme nous l'avons déjà abordé dans la section précédente, nous pouvons compter également le nombre de monômes par degré. Contrairement à l'approche précédente, nous prenons ici tous les monômes - sans distinction des variables qu'ils représentent - nous obtenons alors les graphes des figures 5.51 page 132 et 5.52 page 132.



(a) Vue à 230° (b) Vue à 300°

FIGURE 5.51 – Dénombrement et répartition des degrés des monômes - Processus de chiffrement



(a) Vue à 230° (b) Vue à 300°

FIGURE 5.52 – Dénombrement et répartition des degrés des monômes - Processus de déchiffrement

Nous constatons une différence entre les deux processus. En effet, outre la différence du nombre de monômes déjà évoquée, nous pouvons voir que le nombre de monômes de degré 1, 2, et 6 est quasiment identique. Il en va de même pour les monômes de degré 3 et 5.

5.3.3 Dénombrement des bits de bloc

Chacune des 256 équations booléennes - 128 pour le processus de chiffrement et 128 pour le processus de déchiffrement - que nous avons élaborée peut être

décrite par sa forme algébrique normale. Pour rappel, la forme générale d'une forme algébrique normale est :

$$f(x_0, x_1, \cdots, x_{n-1}) = \sum_{i=(i_0,\cdots,i_{n-1})\in\mathbb{F}_n^2} a_i x_0^{i_0} x_1^{i_1} \cdots x_{n-1}^{i_{n-1}} (mod 2)$$

En fonction de l'équation x_i correspond à un bit de texte clair ou à un bit de chiffré ou à un bit de clef.
Ainsi, à titre d'exemple, l'équation pour le premier bit de la première partie d'un tour de déchiffrement de l'AES - c'est à dire invSubBytes() and invShiftRows() - est la suivante :

$$\begin{aligned}
f(x_0, x_1, \cdots, x_{127}) \;=\; & x_6x_7 + x_5x_6 + x_4 + x_4x_7 + x_4x_5x_7 + x_4x_5x_6 + x_4x_5x_6x_7 + \\
& x_3x_7 + x_3x_6x_7 + x_3x_5 + x_3x_5x_6 + x_3x_5x_6x_7 + x_3x_4 + x_3x_4x_7 + x_3x_4x_6x_7 + x_3x_4x_5x_6 + \\
& x_3x_4x_5x_6x_7 + x_2x_6 + x_2x_5 + x_2x_5x_6 + x_2x_5x_6x_7 + x_2x_4x_6 + x_2x_4x_6x_7 + x_2x_4x_5x_7 + \\
& x_2x_3x_7 + x_2x_3x_6x_7 + x_2x_3x_5x_7 + x_2x_3x_5x_6x_7 + x_2x_3x_4x_6 + x_2x_3x_4x_5x_6 + x_1x_7 + \\
& x_1x_6 + x_1x_6x_7 + x_1x_5 + x_1x_4x_6x_7 + x_1x_4x_5x_7 + x_1x_3x_6 + x_1x_3x_6x_7 + x_1x_3x_5 + \\
& x_1x_3x_5x_6x_7 + x_1x_2 + x_1x_2x_7 + x_1x_2x_6x_7 + x_1x_2x_5x_6x_7 + x_1x_2x_4 + x_1x_2x_4x_7 + \\
& x_1x_2x_4x_6x_7 + x_1x_2x_4x_5x_7 + x_1x_2x_4x_5x_6 + x_1x_2x_4x_5x_6x_7 + x_1x_2x_3x_7 + x_1x_2x_3x_5x_7 + \\
& x_1x_2x_3x_5x_6 + x_1x_2x_3x_4 + x_1x_2x_3x_4x_6 + x_1x_2x_3x_4x_6x_7 + x_1x_2x_3x_4x_5 + x_1x_2x_3x_4x_5x_6 + \\
& x_0x_7 + x_0x_5x_7 + x_0x_5x_6x_7 + x_0x_4x_6x_7 + x_0x_4x_5x_6x_7 + x_0x_3 + x_0x_3x_6 + x_0x_3x_5 + \\
& x_0x_3x_5x_7 + x_0x_3x_5x_6x_7 + x_0x_3x_4x_6x_7 + x_0x_3x_4x_5 + x_0x_3x_4x_5x_7 + x_0x_2x_6 + x_0x_2x_5 + \\
& x_0x_2x_5x_6 + x_0x_2x_5x_6x_7 + x_0x_2x_4 + x_0x_2x_4x_7 + x_0x_2x_4x_6 + x_0x_2x_4x_5 + x_0x_2x_4x_5x_7 + \\
& x_0x_2x_3x_5x_6x_7 + x_0x_2x_3x_4 + x_0x_2x_3x_4x_6x_7 + x_0x_2x_3x_4x_5 + x_0x_2x_3x_4x_5x_6 + x_0x_1x_7 + \\
& x_0x_1x_5x_7 + x_0x_1x_5x_6x_7 + x_0x_1x_4x_7 + x_0x_1x_4x_6x_7 + x_0x_1x_4x_5x_6x_7 + x_0x_1x_3 + x_0x_1x_3x_6 + \\
& x_0x_1x_3x_6x_7 + x_0x_1x_3x_5x_6x_7 + x_0x_1x_3x_4x_5x_7 + x_0x_1x_2x_6x_7 + x_0x_1x_2x_5 + x_0x_1x_2x_5x_7 + \\
& x_0x_1x_2x_4 + x_0x_1x_2x_4x_6 + x_0x_1x_2x_4x_5 + x_0x_1x_2x_4x_5x_7 + x_0x_1x_2x_4x_5x_6 + x_0x_1x_2x_3 + \\
& x_0x_1x_2x_3x_7 + x_0x_1x_2x_3x_5x_7 + x_0x_1x_2x_3x_5x_6 + x_0x_1x_2x_3x_5x_6x_7 + x_0x_1x_2x_3x_4x_5
\end{aligned}$$

Dans cette équation chaque variable x_i correspond à un bit de bloc de clair.
Notre objectif ici est de compter le nombre de fois où chaque bit apparaît dans les équations. Normalement chaque bit devrait apparaître un nombre de fois équivalent. Dans le cas contraire, cela signifierait que certains bits ont moins de poids que d'autres, leur connaissance faciliterait grandement la cryptanalyse.
Afin d'avoir un point de comparaison, nous allons commencer par générer 16 équations booléennes aléatoires à 16 variables. En comptant le nombre de fois où chacune des variables apparaît dans les 16 équations nous obtenons le tableau de la figure 5.53 page 134. Nous y observons une répartition aléatoire des variables sans aucun schéma identifiable.
En appliquant le même mode de comptage aux 128 équations décrivant un tour de chiffrement - `SubBytes()`, `ShiftRows()` et `MixColumns()` -, de déchiffrement - `InvShiftRows()` et `InvSubBytes()` - et d'expansion de la clef de l'AES nous obtenons les graphes des figures 5.54 page 134, 5.55 page 135 et 5.56 page 135. Ces graphes diffèrent d'une utilisation aléatoire de chacune des variables. Dans l'algorithme de l'AES, certains bits de bloc ont plus de poids que d'autres. Ainsi, que ce soit pour le chiffrement que pour le déchiffrement, les bits $x_{7\ (\mathrm{mod}\ 8)}$ sont les plus utilisés et les bits $x_{2\ (\mathrm{mod}\ 8)}$ sont les moins utilisés. Les graphes des figures 5.57 page 136 et 5.58 page 136 présentent l'importance des bits par ordre de poids décroissant.
Le cas de la répartition de l'utilisation des bits dans la fonction d'expansion de la clef est particulièrement intéressant. En effet, nous constatons que seuls les 32 derniers bits sont utilisés. En fait, pour être précis, la répartition détaillée

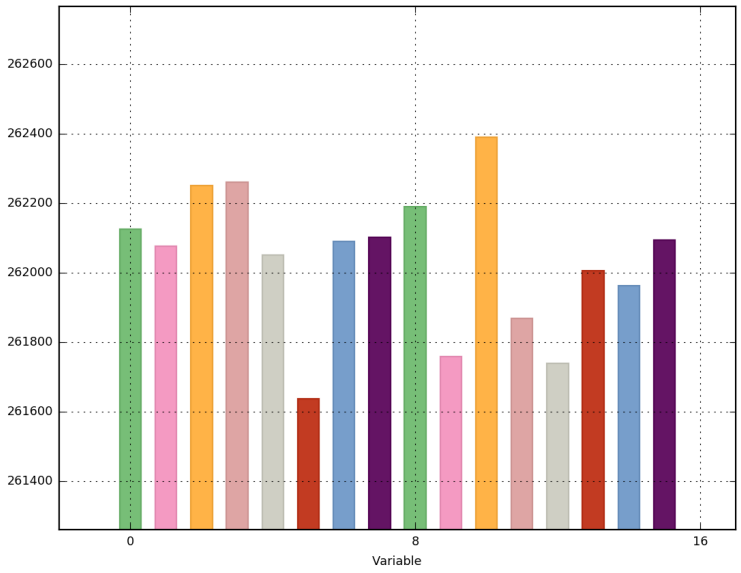


FIGURE 5.53 – Distribution des variables - Fonctions booléennes aléatoires

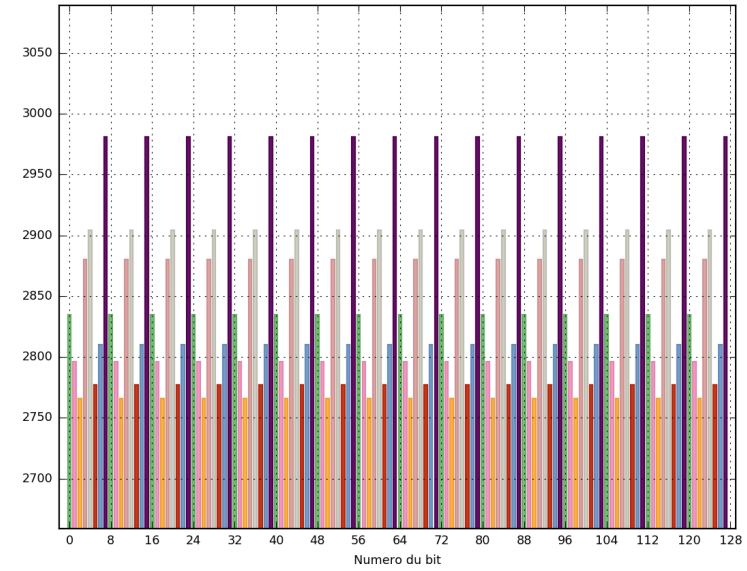


FIGURE 5.54 – Distribution des variables - Fonctions d'un tour de chiffrement

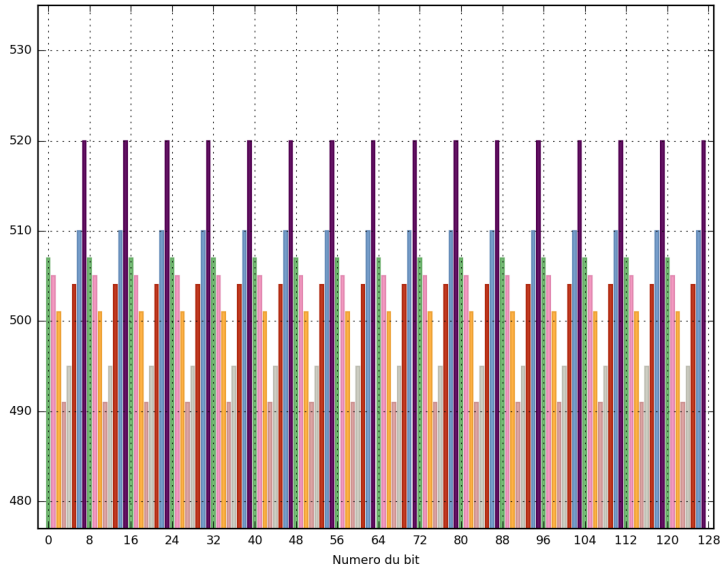


FIGURE 5.55 – Distribution des variables - Fonctions d'un tour de déchiffrement

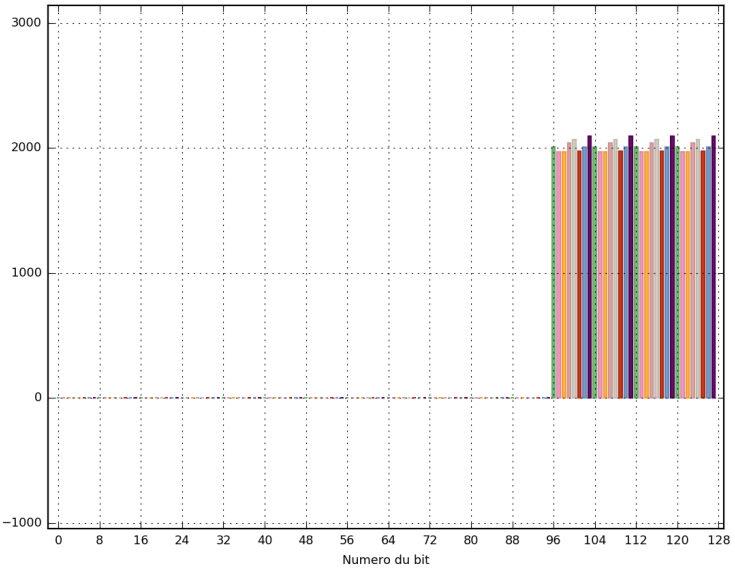


FIGURE 5.56 – Distribution des variables - Fonctions d'expansion de la clef

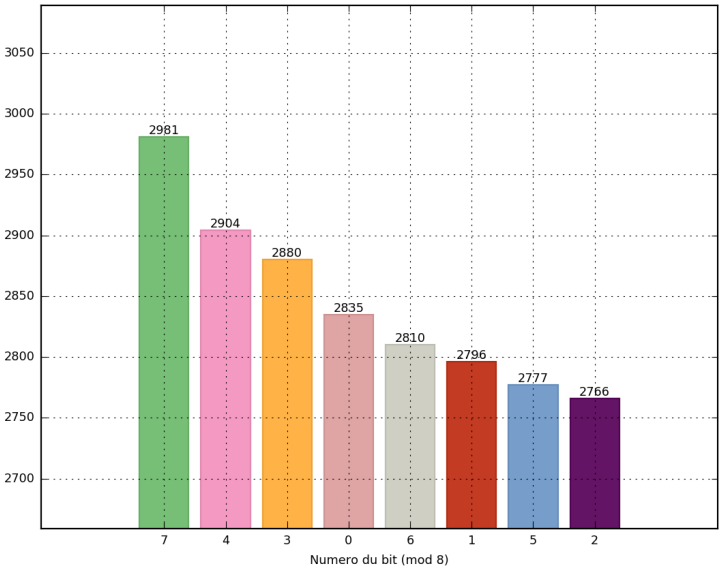


FIGURE 5.57 – Poids des bits - Fonctions d'un tour de chiffrement

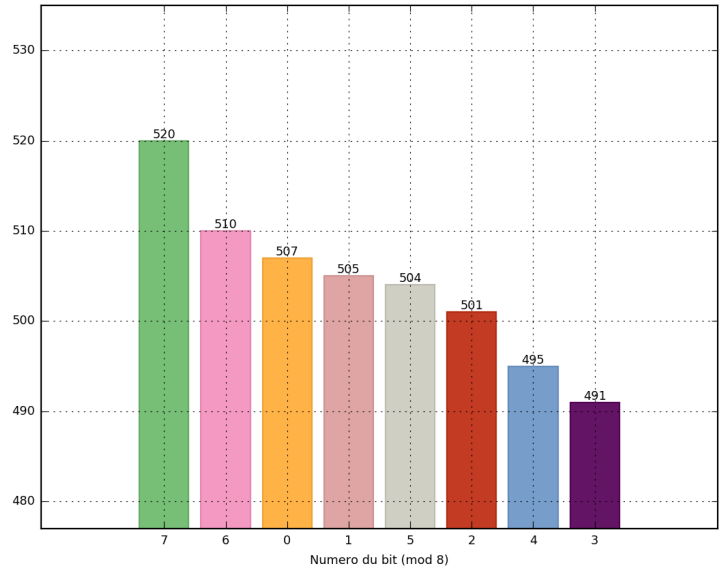


FIGURE 5.58 – Poids des bits - Fonctions d'un tour de déchiffrement

de l'utilisation des bits de clefs est présentée dans le tableau de la figure 5.59 page 137. Nous pouvons voir que, dans le processus d'expansion de la clef, les 32 premiers bits de clefs sont utilisés quatre fois, les 32 bits suivants trois fois, les 32 bits suivants seulement deux fois et que seuls les 32 derniers bits sont réellement utilisés. Cela signifie que sur 128 bits de clefs, les 32 derniers bits ont plus de poids. Cette vulnérabilité est compensée d'une part par l'intrication des bits de clef dans le processus d'expansion de la clef et par le jeu des tours.

	Numéro du bit							
Numéro de l'octet	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	4	4	4	4	4	4	4	4
2	4	4	4	4	4	4	4	4
3	4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4	4
5	3	3	3	3	3	3	3	3
6	3	3	3	3	3	3	3	3
7	3	3	3	3	3	3	3	3
8	3	3	3	3	3	3	3	3
9	2	2	2	2	2	2	2	2
10	2	2	2	2	2	2	2	2
11	2	2	2	2	2	2	2	2
12	2	2	2	2	2	2	2	2
13	2005	1969	1969	2041	2065	1973	2009	2093
14	2005	1969	1969	2041	2065	1973	2009	2093
15	2005	1969	1969	2041	2065	1973	2009	2093
16	2005	1969	1969	2041	2065	1973	2009	2093

FIGURE 5.59 – Utilisation des bits de la clef de tour

Afin de mieux visualiser l'impact de ce biais nous allons utiliser la même approche que celle présentée pour les PRBG. Pour cela nous utilisons l'AES pour chiffrer une séquence de blocs en fixant tous les bits à 0 avec une clef de chiffrement dont les 96 premiers bits sont fixés à 0 et dont les 32 derniers bits s'incrémentent. Nous obtenons ainsi une séquence de blocs que nous pouvons ensuite visualiser dans l'environnement 3D présenté plus haut. Le tableau 5.60 montre le résultat obtenu pour les premiers éléments de la séquence en utilisant un tour et le tableau 5.61 le résultat sur deux tours de l'AES.

Bloc de clef	Bloc de chiffré
00000000000000000000000000000000	01000000010000000100000001000000
00000001000000000000000000000000	010000011e1f213f0100000101000001
00000002000000000000000000000000	0100000215143c2a0100000201000002
00000003000000000000000000000000	0100000319182833010000301000003
00000004000000000000000000000000	010000049091a83d0100000401000004
00000005000000000000000000000000	0100000509081815010000501000005
00000006000000000000000000000000	010000060d0c141e0100000601000006
00000007000000000000000000000000	01000007a7a6f150010000701000007

FIGURE 5.60 – Blocs de chiffrés obtenus sur 1 tour de l'AES

Dans notre environement en trois dimensions, sur un échantillon comprenant

Bloc de clef	Bloc de chiffré
00000000000000000000000000000000	c6e4e48ba48787e8c6e4e48ba48787e8
00000001000000000000000000000000	27a63717a796f9c4d0f28fa62521d048
00000002000000000000000000000000	62d95538fab68aaf4062619e3810a448
00000003000000000000000000000000	977b2976f7372c3b664434d3eb662a8f
00000004000000000000000000000000	6aaad0540d0a4df582a0560794ee0870
00000005000000000000000000000000	3b4a67cd56f22a7ffcde62fa62c6d036
00000006000000000000000000000000	76c9db08e5208e16d7f561af313b0c69
00000007000000000000000000000000	a9ccea9c4201eedff6d40cecc07c517d

FIGURE 5.61 – Blocs de chiffrés obtenus sur 2 tours de l'AES

les 2^{20} premières séquences de clefs, nous obtenons la figure 5.62 sur un tour de l'AES et la figure 5.63 sur deux tours de l'AES. En comparant ces deux figures on constate que la différence de poids des bits de clefs dans l'algorithme d'expansion de la clef s'estompe dès le deuxième tour.

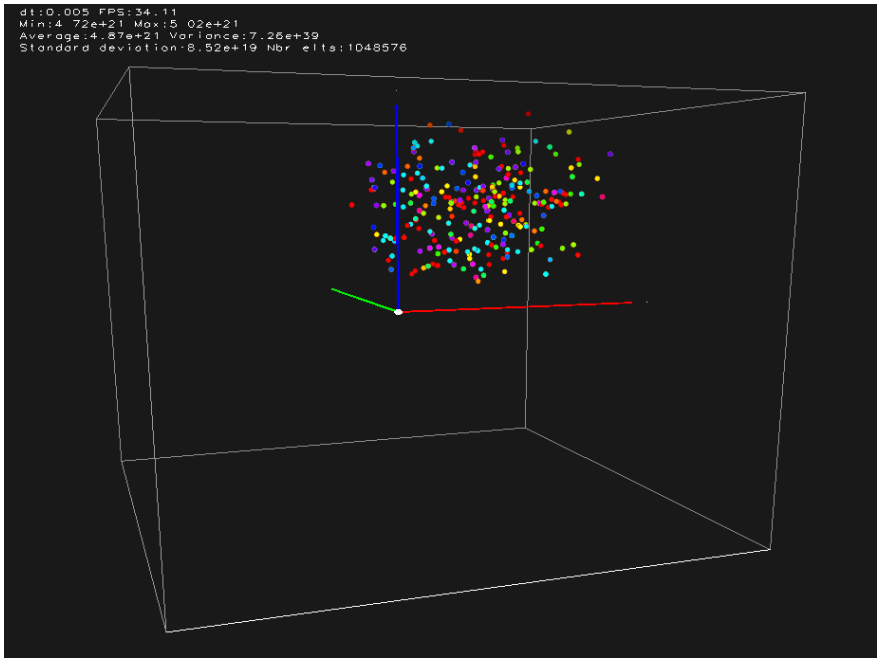


FIGURE 5.62 – Séquence obtenue sur 1 tour de l'AES

5.3.4 Dénombrement des bits de bloc deux par deux

Dans cette dernière étape de notre analyse combinatoire, nous allons dénombrer les paires de bits.
Toujours en partant de la forme algébrique normale des équations booléennes que nous avons générées, nous allons compter l'utilisation des bits deux à deux. Par exemple, si nous prenons le début de l'équation du premier bit d'un tour

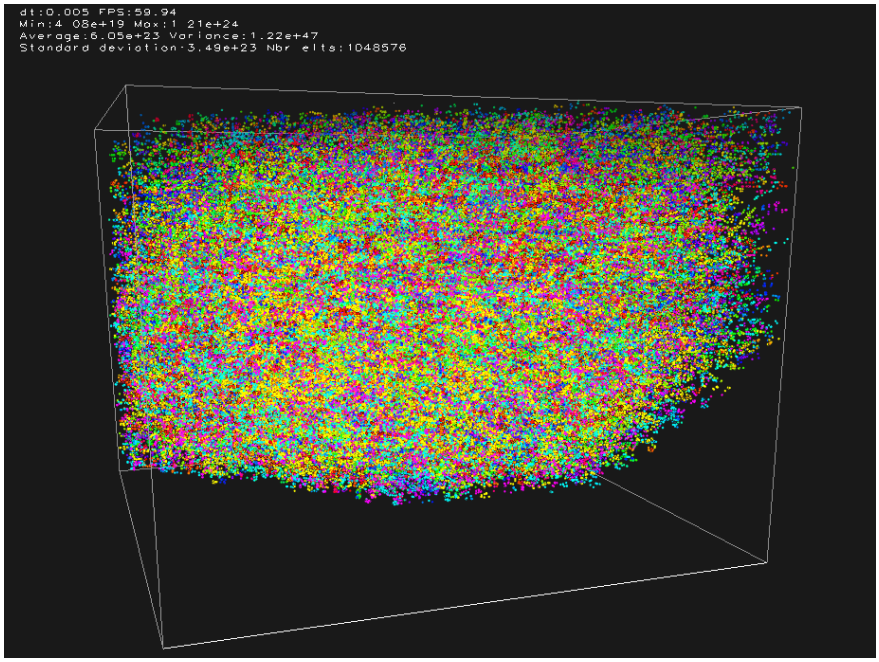


FIGURE 5.63 – Séquence obtenue sur 2 tours de l'AES

de déchiffrement :

$$f(b_0, b_1, \cdots, b_{127}) = b_6b_7 + b_5b_6 + b_4 + b_4b_7 + b_4b_5b_7 + b_4b_5b_6 + b_4b_5b_6b_7$$

nous avons les paires de bits suivantes qui apparaissent : $(6,7)$, $(5,6)$, $(4,7)$, $(4,5)$, $(5,7)$, $(4,5)$, $(5,6)$, $(4,5)$, $(5,6)$, $(6,7)$. Soit deux fois la paire $(6,7)$, trois fois la paire $(5,6)$, une fois la paire $(4,7)$, trois fois la paire $(4,5)$ et une fois la paire $(5,7)$.
Afin d'avoir un point de comparaison, nous générons 16 équations booléennes aléatoires à 16 variables et nous comptons la répartition des paires de variables. Nous obtenons le graphe de la figure 5.64 page 140. Sur ce graphe, chaque paire est représentée par une colonne placée au croisement des indices correspondants. La hauteur de la colonne est donnée par le nombre de fois où chaque paire apparaît dans les équations.
En appliquant le même mode de comptage aux 128 équations décrivant un tour de chiffrement, de déchiffrement et d'expansion de la clef de l'AES nous obtenons les graphes des figures 5.65 page 140, 5.66 page 141 et 5.67 page 141.
Ces trois graphes confirment les constats effectués précédemment. Pour les fonctions de tour ainsi que pour la fonction d'expansion de la clef, il n'y a pas de répartition aléatoire des paires de bits. Toutes les paires sont placées sur la médiane. Enfin, pour la fonction d'expansion de la clef, nous retrouvons l'utilisation des 32 derniers bits essentiellement.

—=oOo=—

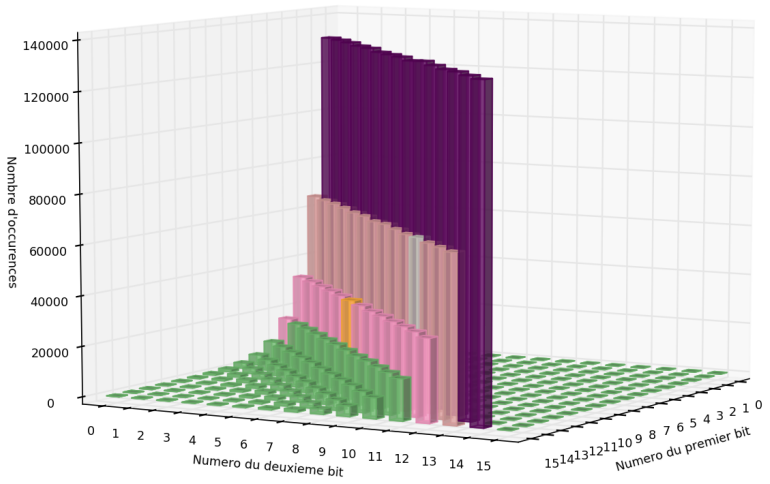


FIGURE 5.64 – Distribution des paires de variables - Fonctions booléennes aléatoires

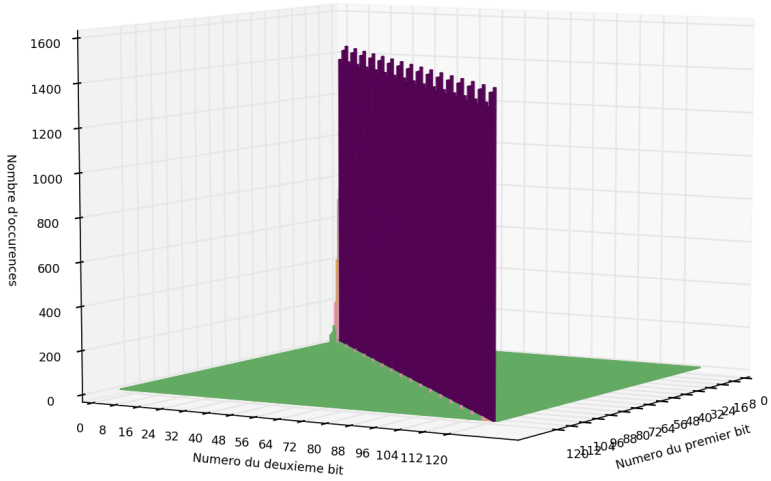


FIGURE 5.65 – Distribution des variables 2 à 2 - Fonctions d'un tour de chiffrement

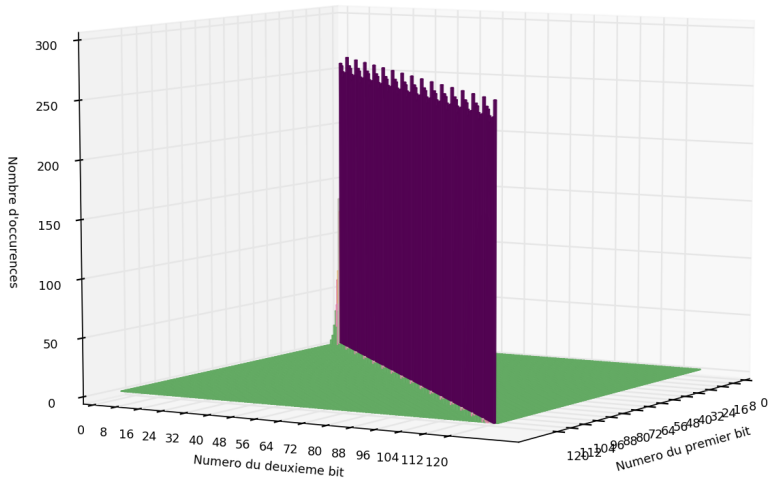


FIGURE 5.66 – Distribution des variables 2 à 2 - Fonctions d'un tour de déchiffrement

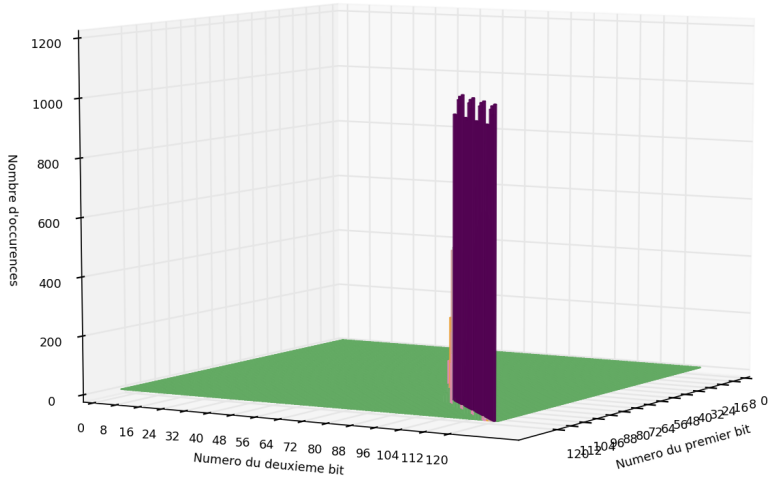


FIGURE 5.67 – Distribution des variables 2 à 2 - Fonctions d'expansion de la clef

Notre analyse des formes algébriques normales des fonctions booléennes décrivant le processus de chiffrement du mini-AES et de l'AES, à l'aide du test statistique de Möbius, nous a permis de trouver des irrégularités dans le mini-AES et surtout nous a permis de montrer formellement que la S-Box de l'AES ne présente pas de biais statistique. En revanche, toujours selon le test statistique de Möbius, nous avons montré que la fonction de tour de chiffrement de l'AES présente des biais statistiques alors que la fonction de tour de déchiffrement est conforme.
Enfin, notre analyse combinatoire montre une forte disparité entre les équations de tour du chiffrement et du déchiffrement. Surtout, cette analyse montre l'importance des 32 derniers bits de clefs par rapport au 96 premiers. Nous avons vu que, sur un petit échantillon, cette disparité ne semble pas avoir de conséquences. Cependant, une telle différence dans le poids des bits de clef n'est pas normale et ouvre une nouvelle perspective de recherche pour l'exploiter en termes de cryptanalyse.