La Génération d'Aléas sur Ordinateur

Louis Goubin
CP8 Crypto Lab
SchlumbergerSema
36-38 rue Princesse – BP45
78430 Louveciennes Cedex
LGoubin@slb.com

Jacques Patarin
Laboratoire PRiSM
Université de Versailles
45 avenue des Etats-Unis
78035 Versailles Cedex
jap@prism.uvsq.fr

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." (John von Neumann, 1951)

"Random numbers should not be generated with a method chosen at random. Some theory should be used." (Donald Knuth, 1969)

1 Introduction

Les nombres "aléatoires" interviennent dans de très nombreux domaines parmi lesquels on trouve les simulations numériques de phénomènes physiques (ce sont les méthodes dites de "Monte Carlo"), le prélèvement d'échantillons au hasard, l'analyse numérique, les tests de détection de défauts de composants informatiques, la génération de clés secrètes pour la cryptographie, la création d'art graphique (génération aléatoire de couleurs ou de formes par exemple), ou encore la programmation des jeux de casino ou de loterie.

De plus, certains algorithmes utilisant les nombres aléatoires (ces algorithmes sont dits "probabilistes") se révèlent plus performants que tous les algorithmes connus n'utilisant pas de nombres aléatoires (qui sont dits "déterministes"). Il existe même des problèmes résolubles par des algorithmes probabilistes et pour lesquels on sait qu'il n'existe pas de solution déterministe (cf [4] ou [3]). Ainsi le problème de la génération de nombres aléatoires est un problème très important et très utile. Certains en viennent même à considérer que les nombres aléatoires ont une certaine "valeur marchande" : certaines compagnies (comme la "Rand Corporation" aux USA) ont ainsi édité des tables de plusieurs millions de nombres "aléatoires", dans les années 50. De nos jours, les "nombres aléatoires" sont engendrés soit par des moyens physiques, soit par des programmes sur ordinateur, soit en combinant les deux.

Le problème de la génération de nombres "aléatoires" et suivant une loi de distribution donnée se ramène en fait généralement au problème de la génération de bits "aléatoires" suivant une distribution uniforme, c'est-à-dire d'une suite de valeurs valant 0 ou 1, les 0 et les 1 apparaissant comme tirés indépendamment les uns des autres, et avec une distribution uniforme.

On parle de nombres "réellement aléatoires" lorsqu'ils sont engendrés par des processus physiques indépendamment les uns des autres et avec une distribution uniforme. Par exemple en tirant n fois à pile ou face, sans "tricher" et avec une pièce "normale", on considère que l'on engendre n bits réellement aléatoires. PGP (Pretty Good Privacy), un célèbre logiciel de cryptographie sur Internet, mesure en millisecondes l'instant de la frappe de touches au clavier, pour essayer d'engendrer de tels nombres "réellement aléatoires". Il est à noter que la génération physique de nombres "réellement aléatoires" est souvent délicate, particulièrement quand une grande quantité de ces nombres est requise.

En revanche, on parle de nombres "pseudo-aléatoires" lorsqu'ils sont engendrés de façon déterministe par un programme informatique, ce programme utilisant éventuellement une valeur secrète K (la taille de K étant inférieure au nombre de bits que le programme sait engendrer). L'idée est la suivante : le générateur sera un bon générateur de nombres pseudo-aléatoires si on ne parvient pas à distinguer les bits qu'il engendre de bits réellement aléatoires.

En fait, la réalisation de bons générateurs de nombres pseudo-aléatoires est un problème difficile. Comme on le voit dans la citation de D. Knuth au début de cet article, ce n'est certainement pas en écrivant un code informatique "n'importe comment" que l'on obtient un bon générateur de nombres pseudo-aléatoires. Tout au contraire, de tels générateurs ne sont obtenus qu'avec grand soin et en introduisant des structures très précises dans le programme.

L'introduction des bonnes définitions mathématiques (qu'est-ce qu'un nombre aléatoire ? que signifie "ne pas parvenir à distinguer une suite de bits donnée d'une suite de bits réellement aléatoire" ?) s'est aussi révélée être un problème difficile : ce n'est qu'au début des années 80 que les définitions actuelles (à la fois précises et correspondant aux besoins) furent introduites.

On peut noter dès maintenant qu'il existe des liens profonds entre la théorie des générateurs de nombres pseudo-aléatoires et la cryptographie à clé secrète. On peut d'ailleurs, sans vraiment exagérer, considérer que ce sont deux noms différents pour la même théorie. En effet, on demande aux algorithmes à clé secrète de savoir chiffrer, à partir d'une petite clé K, de nombreux messages, de telle sorte que les messages chiffrés apparaissent indistinguables de suites de bits aléatoires lorsque K n'est pas connue. Ainsi chaque algorithme de chiffrement à clé secrète peut facilement être utilisé pour engendrer des nombres aléatoires. Réciproquement, à partir d'un bon générateur de bits pseudo-aléatoires, on sait comment réaliser relativement facilement des fonctions de chiffrement à clé secrète (voir [10] ou [11] pour le chiffrement par flot, et [9] pour le chiffrement par blocs).

Certes, pour de nombreuses applications (physiques ou esthétiques par exemple), l'utilisation de nombres "relativement aléatoires" est souvent suffisante. Ici, par "nombres relativement aléatoires", nous entendons des nombres qui peuvent éventuellement être distingués de nombres réellement aléatoires, mais tels que ces distinctions soient complexes et n'induisent pas de modification sensible pour les applications visées. En revanche, pour d'autres applications (en particulier pour des applications liées à la sécurité), il est important d'avoir réellement des nombres pseudo-aléatoires indistinguables de nombres réellement aléatoires. Notons que les tables publiées de nombres aléatoires sont alors en général inutilisables, puisque les ennemis potentiels peuvent les connaître.

L'objet de cet article sera donc d'une part d'exposer les méthodes actuellement utilisées pour engendrer des nombres "aléatoires" sur ordinateur, et d'autre part de présenter les notions et le formalisme mathématique qui ont été développés pour étudier ce problème.

2 Exemples de générateurs à base de théorie des nombres

Un générateur pseudo-aléatoire est une fonction f qui prend en entrée une chaîne courte s appelée graine, et donne en sortie une chaîne $f(s)$ plus longue, et qui "ressemble" à une chaîne réellement aléatoire.
La définition précise (que nous verrons au paragraphe 3) est assez complexe, et on va d'abord essayer d'en donner une notion intuitive sur quelques exemples.

2.1 Le générateur congruentiel linéaire

La graine est un nombre x_0 tel que $0 \leq x_0 \leq M-1$. Le générateur produit une suite (x_n) d'entiers tels que $\forall n$, $0 \leq x_n \leq M-1$, définie par la relation de récurrence suivante :

$$\forall n \in \mathbf{N},\ x_{n+1} \equiv a x_n + b \bmod M,$$

où a, b et M sont les 3 paramètres décrivant le générateur.
Il est évident que la suite ainsi produite est périodique, avec une période ne pouvant excéder M (c'est le cas plus généralement de toute suite (x_n) définie par une relation de la forme $x_{n+1} \equiv f(x_n) \bmod M$). Toutefois, on peut choisir les paramètres a, b et M de manière à obtenir une période maximale, *i.e.* égale à M :

Theorème 2.1 (Hull & Dobell, 1962) *La période est de longueur M si et seulement si :*
(i) b est premier avec M.
(ii) Tout facteur premier de M est aussi facteur de $a-1$.
(iii) Si M est divisible par 4, alors $a-1$ aussi.

Le générateur congruentiel linéaire n'est cependant pas acceptable comme générateur pseudo-aléatoire, à cause du résultat suivant :

Theorème 2.2 (Boyar, 1989) *Il existe un algorithme en temps polynomial qui, prenant en entrée les valeurs x_1, ..., x_n données par un générateur congruentiel linéaire (modulo M), donne une prédiction $\hat{x}_{n+1}$. Cet algorithme a la propriété que s'il est utilisé successivement pour $n = 1, 2, 3, ...$, il commet au plus $\log M + 3$ erreurs.*

Par conséquent, les générateurs congruentiels linéaires ne satisfont pas la notion intuitive de "non-prédictabilité" des nombres engendrés, et ne sont donc pas recommandables pour des applications sensibles.

2.2 Le générateur $1/p$

Les paramètres de ce générateur sont : un nombre premier $p \geq 3$, un générateur g de $(\mathbf{Z}/p\mathbf{Z})^*$ (le groupe des éléments inversibles modulo p), et un nombre entier r tel que $0 < r < g$.
La suite (x_n) produite par le générateur est définie par l'écriture en base g du nombre rationnel $\frac{r}{p}$:

$$\frac{r}{p} = 0, x_1 x_2 x_3 ...$$

Comme dans le cas du générateur congruentiel linéaire, la suite obtenue est nécessairement périodique, et ici sa période est au plus $p - 1$.
On peut en outre choisir les paramètres de façon à atteindre cette période $p - 1$, et le théorème suivant montre que la suite engendrée vérifie des propriétés encore plus fortes :

Theorème 2.3 *Pour toute racine primitive g modulo p, et pour tout entier r tel que $0 < r < g$, la suite (x_n) produite par le générateur $1/p$ est une suite de de Bruijn de période $p - 1$ et de base g, c'est-à-dire :*

(i) La suite (x_n) est $(p-1)$-périodique.

(ii) Toute suite finie à valeurs dans $\{0, ..., g-1\}$ et de longueur $|p| - 1$ apparaît au moins une fois dans la suite (x_n) (ici $|p|$ désigne le nombre de chiffres de p en base g).

(iii) Toute suite finie à valeurs dans $\{0, ..., g-1\}$ et de longueur $|p|$ apparaît au plus une fois dans la suite (x_n).

Malheureusement, le générateur $1/p$ est encore prédictible : en utilisant des développements en fraction continue, on peut montrer le résultat suivant :

Theorème 2.4 (Blum, Blum & Shub, 1986) *(i) Étant donnés $\log_g p + 1$ termes consécutifs de la suite produite par une générateur $1/p$, on peut prédire le terme suivant en temps polynomial en $\log_g p$. (ii) Étant donnés $2 \log_g p + 3$ termes consécutifs d'une telle suite, on peut trouver p en temps polynomial en $\log_g p$.*

2.3 Un générateur à base de registres à décalage linéaires

Un registre à décalage linéaire de degré k est un algorithme qui produit une suite de bits de la manière suivante. Le vecteur $(z_1, ..., z_k)$ est utilisé pour initialiser le registre à décalage $(m_1, ..., m_k)$. À chaque étape de l'algorithme, on effectue les opérations suivantes :

1. m_1 donne le terme suivant de la suite (z_n).
2. m_2, ..., m_k sont décalés d'un cran vers la gauche.
3. La nouvelle valeur de m_k est donnée par : $\sum_{j=0}^{k-1} c_j m_{j+1} \bmod 2$, où les coefficients c_j valent 0 ou 1.

Ce procédé revient à définir la suite (z_n) par la formule de récurrence linéaire suivante :

$$\forall n \in \mathbf{N}^*,\ z_{n+k} \equiv \sum_{j=0}^{k-1} c_j z_{n+j} \bmod 2.$$

Un registre à décalage linéaire peut être vu comme un générateur de bits à partir de la graine $(z_1, ..., z_k)$. Étant donnée une graine de k bits, un registre à décalage linéaire de degré k engendre une suite de période au plus $2^k - 1$. Comme dans le cas des générateurs congruentiels linéaires, les coefficients c_j peuvent être choisis de manière à atteindre cette période maximale $2^k - 1$. Cependant, les générateurs de bits construits à partir d'un registre à décalage linéaire ne sont pas sûrs, car on peut montrer que la connaissance de $2k$ termes consécutifs de la suite (z_n) suffit pour reconstituer la graine, et par conséquent pour recalculer tous les termes de la suite. La propriété intuitive de "non-prédictabilité" fait à nouveau défaut, et on n'obtient donc pas un générateur pseudo-aléatoire.

Une idée a néanmoins été avancée par Coppersmith, Krawczyk et Mansour (cf [1]) pour construire un candidat de générateur pseudo-aléatoire de bits, appelé *générateur réducteur*. On suppose que l'on dispose de deux registres à décalage linéaires de degrés respectifs k_1 et k_2. La graine, constituée de $k_1 + k_2$ bits, sert à initialiser les deux registres. Le premier registre engendre une suite a_1, a_2, ... De même, le second registre engendre une suite b_1, b_2, ... On construit alors la suite z_1, z_2, ... en posant $z_n = a_{i_n}$, où i_n est la position du n-ième 1 dans la suite b_1, b_2, ... Cette méthode conduit à un candidat de générateur pseudo-aléatoire de bits très rapide.

2.4 Les générateurs à base de puissance modulaire

La graine est un entier x_0 tel que $0 \leq x_0 \leq N - 1$, et on engendre un suite (x_n) grâce à la formule de récurrence suivante :

$$x_{n+1} \equiv (x_n)^e \bmod N.$$

Deux cas prennent une importance particulière :

2.4.1 Le générateur RSA

Il a été étudié par Alexi, Chor, Goldreich et Schnorr en 1988.
On suppose que N est le produit de deux nombres premiers impairs p et q, et que e est premier avec $\varphi(N) = (p-1)(q-1)$ (l'indicatrice d'Euler de N).
Il est facile de voir que l'application $x \mapsto x^e \bmod N$ est alors bijective. Les formules suivantes :

$$\forall n \in \mathbf{N},\ \begin{cases} x_{n+1} \equiv (x_n)^e \bmod N \\ z_{n+1} \equiv x_{n+1} \bmod 2 \end{cases}$$

avec $\forall n \in \mathbf{N}$, $0 \leq x_n \leq N - 1$ et $\forall n \in \mathbf{N}^*$, $z_n \in \{0, 1\}$, définissent alors un candidat de générateur pseudo-aléatoire de bits, appelé générateur RSA, en référence au célèbre algorithme cryptographique du même nom, dont la fonction de chiffrement est aussi $x \mapsto x^e \bmod N$.

2.4.2 Les générateurs quadratiques

On suppose également $N = pq$, où p et q sont deux nombres premiers tels que $p \equiv q \equiv 3 \bmod 4$, et on choisit ici $d = 2$. On note $Q(N)$ l'ensemble des *résidus quadratiques* modulo N, défini par :

$$Q(N) = \{y \bmod N,\ (y, N) = 1 \text{ et } \exists x,\ y \equiv x^2 \bmod N\},$$

L'application $x \mapsto x^2 \bmod N$ n'est pas bijective (un élément de $Q(N)$ possède en effet 4 antécédents). Néanmoins, si on se restreint aux résidus quadratiques, alors on retrouve une bijection.
Ici, la suite produite par le générateur est définie par :

$$\forall n \in \mathbf{N},\ \begin{cases} x_{n+1} \equiv (x_n)^2 \bmod N \\ z_{n+1} \equiv x_{n+1} \bmod 2 \end{cases}$$

avec $\forall n \in \mathbf{N}^*$, $z_n \in \{0, 1\}$.
Il existe plusieurs variantes selon le choix de l'intervalle où sont choisis les représentants x_n modulo N. Deux exemples ont été particulièrement étudiés :

- Le *générateur BBS* : proposé par Blum, Blum et Shub en 1986, il correspond au choix $x_n \in [0, N[$.
- Le *générateur quadratique centré* : étudié par Alexi, Chor, Goldreich et Schnorr en 1988, il correspond au choix $x_n \in]-\frac{N}{2}, \frac{N}{2}[$.

Ce sont deux candidats différents de générateurs pseudo-aléatoires de bits.

2.5 Un générateur à base d'exponentielle discrète

Le générateur est défini par un nombre premier $p \geq 3$ et un générateur g de $(\mathbf{Z}/p\mathbf{Z})^*$. La graine est un élément x_0 de $(\mathbf{Z}/p\mathbf{Z})^*$. La suite de bits (z_n) produite par le générateur est alors définie de la manière suivante :

$$\forall n \in \mathbf{N}, \quad \begin{cases} x_{n+1} \equiv g^{x_n} \mod p \\ z_{n+1} = \begin{cases} 1 & \text{si } x_{n+1} > \frac{p}{2} \\ 0 & \text{si } x_{n+1} < \frac{p}{2} \end{cases} \end{cases}$$

avec $\forall n \in \mathbf{N}$, $0 \leq x_n \leq p-1$. Ce candidat de générateur pseudo-aléatoire de bits a été proposé par Blum et Micali en 1984.

3 Générateurs à base de chiffrement par bloc ou de hachage

3.1 Génération avec le chiffrement par blocs TDES

Dans beaucoup de cas pratiques, la génération de nombres pseudo-aléatoires se fait en utilisant des outils classiques de la cryptographie à clé secrète comme un algorithme de chiffrement par bloc ou une fonction de hachage.
En effet, dans beaucoup d'applications, ces algorithmes sont déjà présents (dans les cartes à puce par exemple) et ils sont très faciles à utiliser. Il est donc souvent commode de les utiliser plutôt que d'ajouter un algorithme supplémentaire spécialement dédié à la génération d'aléas.
Lálgorithme à clé secrète le plus utilisé actuellement, et qui offre apparemment un très grand niveau de sécurité (ce n'est pas prouvé mais le contraire serait une immense surprise tant les attaques connues sont loin d'être réalistes) est le TDES (ou Triple-DES). On pourra trouver une description de ce schéma dans tout livre général de cryptographie, [11] par exemple. Le TDES prend une entrée x de 64 bits, donne une sortie y de 64 bits, et dépend d'une clé secrète k de 112 bits.
De plus cette fonction est bijective : lorsque la clé k est connue et fixée, on peut calculer $y = \mathrm{TDES}_k(x)$ et $x = \mathrm{TDES}_k^{-1}(y)$.
Voici comment on peut utiliser le TDES pour générer des bits pseudo-aléatoires (par exemple).

1. Générer une clé k de façon réellement aléatoire (en tirant 64 fois à pile ou face par exemple), et mettre le compteur r à zéro.
2. Calculer $y = \mathrm{TDES}_k(r)$. La valeur y fournit 64 bits pseudo-aléatoires.
3. Incrémenter r d'une unité et reprendre l'étape 2.

On peut recommencer ainsi des milliers de fois.

Remarque: Au-delà de 2^{32} fois, le "paradoxe des anniversaires" peut permettre de distinguer une chaîne de plus de 2^{32} bits générés ainsi d'une chaîne de bits réellement aléatoires.

3.2 Génération avec le chiffrement par bloc AES

Récemment, un nouvel algorithme de chiffrement par bloc, nommé AES, a été normalisé. Il prend en entrée une valeur x de 128 bits, donne une sortie y de 128 bits et dépend d'une clé k de 128 bits également (voir par exemple [2] pour une description de cet algorithme). Comme précédemment, on peut utiliser cet algorithme pour générer des bits pseudo-aléatoires, par exemple ainsi :

1. Générer une clé k de façon réellement aléatoire (en tirant 128 fois à pile ou face par exemple), et mettre le compteur r à zéro.

2. Calculer $y = \text{AES}_k(r)$. La valeur y fournit 128 bits pseudo-aléatoires.

3. Incrémenter r d'une unité et reprendre l'étape 2.

On peut recommencer ainsi des milliards et des milliards de fois, la chaîne des bits générés ne sera pas distinguable de bits réellement aléatoires (si l'AES est un bon algorithme, ce que l'on estime en général, mais que l'on ne sait pas prouver).

Remarque: C'est au-delà de 2^{64} fois environ que le "paradoxe des anniversaires" permet ici de distinguer les bits générés de bits réellement aléatoires. Ce chiffre est si grand qu'en pratique ce n'est pas une réelle restriction.

3.3 Génération avec une fonction de hachage

La fonction SHA-1 est la "fonction de hachage" couramment utilisée actuellement. Elle prend une entrée x de taille quelconque et donne une sortie $y = \text{SHA-1}(x)$ de 160 bits. (Une description de SHA-1 se trouve dans [11] par exemple.)
Il est possible de générer des bits pseudo-aléatoires à partir de SHA-1 par exemple ainsi :

1. Générer une clé k de façon réellement aléatoire, de 80 bits environ (par exemple en tirant 80 fois à pile ou face), et mettre le compteur r à zéro.

2. Calculer $y = \text{SHA-1}(k||r)$ (ici $||$ désigne la concaténation).

3. Incrémenter r d'une unité et reprendre l'étape 2.

On peut itérer ce procédé sans souci, car il n'y a pas de paradoxe des anniversaires ici, puisque SHA-1 n'est pas bijective.
On ne connaît pas d'attaque réaliste sur une telle génération de bits. Cependant, on n'a pas non plus de preuve de sécurité.

4 L'approche théorique

La notion de "générateur pseudo-aléatoire de bits" est actuellement définie dans le cas dit "uniforme", ou dans le cas dit "non uniforme". Le cas "non uniforme" est peut-être le plus important, car il prend en compte des attaques un peu plus générales. Voici ces deux définitions.
On note I_n l'ensemble de toutes les chaînes de n bits, c'est-à-dire $I_n = \{0,1\}^n$. Un "algorithme de calcul" représente un programme informatique (écrit en C ou en Pascal par exemple) dont le listing contenant le code des instructions est toujours fini et fixé (si on veut on peut aussi appeler "mémoire morte" ce code). L'algorithme de calcul prend en entrée une ou plusieurs valeurs qu'on lui fournira, puis exécute les instructions de son code, et éventuellement s'arrête en donnant une ou plusieurs valeurs de sortie. Il peut arriver que pour certaines entrées, le programme fasse des calculs indéfiniment, par exemple s'il y a des instructions sur lesquelles il boucle indéfiniment. Notons que, lors de l'exécution du programme, celui-ci peut avoir accès à toute la mémoire (vive) qu'il désire.

Définition 1 : cas uniforme
Un "générateur pseudo-aléatoire de bits" est une suite de fonctions $f_n : I_n \to I_{\ell(n)}$, où $\ell(n)$ est un polynôme en n, avec $\ell(n) \geq n+1$, tel que :

1. Il existe un polynôme $P(n)$ et il existe un algorithme de calcul W prenant un entier n et un élément x de I_n en entrée, tel que quelles que soient ces deux entrées n et x, W calcule la valeur $f_n(x)$ en moins de $P(n)$ opérations élémentaires.

2. Pour tous polynômes $Q(n)$ et $P(n)$, et pour tout algorithme de calcul T prenant un entier n et un élément A de $I_{\ell(n)}$ en entrée et donnant 0 ou 1 en sortie en moins de $P(n)$ opérations élémentaires, on a :
$$\exists n_0 \in \mathbf{N}, \text{ tel que } \forall n \geq n_0, \ |P_1 - P_1^*| < \frac{1}{Q(n)},$$

où P_1^* est la probabilité que T donne 1 en sortie lorsque A est tirée aléatoirement dans $I_{\ell(n)}$, et P_1 la probabilité que T donne 1 en sortie lorsqu'un élément a est tiré aléatoirement dans I_n et que l'on prend $A = f_n(a)$.

Pour résumer, un générateur pseudo-aléatoire de bits est donc une technique pour étendre en temps polynomial en n une suite réellement aléatoire de n bits en une suite pseudo-aléatoire de $\ell(n)$ bits. Si a est choisi aléatoirement dans I_n, alors $f_n(a)$ peut être calculé en temps polynomial, mais ne peut pas être distingué (en temps polynomial, et avec une bonne probabilité) d'une suite réellement aléatoire de $I_{\ell(n)}$.

Définition 2 : cas non uniforme
Un "générateur pseudo-aléatoire de bits" est une suite de fonctions $f_n : I_n \to I_{\ell(n)}$, où $\ell(n)$ est un polynôme en n, avec $\ell(n) \geq n$, tel que :

1. Il existe un polynôme $P(n)$ et – pour tout $n \in \mathbf{N}$ – un algorithme de calcul W_n prenant un élément x de I_n en entrée tel que :
   - Le nombre total d'instructions du code de W_n est $\leq P(n)$.
   - Quelle que soit l'entrée x, W_n calcule la valeur $f_n(x)$ en moins de $P(n)$ opérations élémentaires.
2. Soit $(T_n)_{n\in\mathbf{N}}$ une suite d'algorithmes de calcul tels qu'il existe un polynôme P tel que :
   - Pour tout $n \in \mathbf{N}$, le nombre total d'instructions du code de T_n est $\leq P(n)$.
   - Pour tout $n \in \mathbf{N}$, T_n prend un élément A de $\{0,1\}^{\ell(n)}$ en entrée et donne 0 ou 1 en sortie en moins de $P(n)$ opérations élémentaires.

   Alors, pour tout polynôme $Q(n)$ et pour toute telle suite d'algorithmes $(T_n)_{n\in\mathbf{N}}$, on a :

$$\exists n_0 \in \mathbf{N}, \text{ tel que } \forall n \geq n_0, \ |P_1 - P_1^*| < \frac{1}{Q(n)},$$

   où P_1^* est la probabilité que T donne 1 en sortie lorsque A est tirée aléatoirement dans $I_{\ell(n)}$, et P_1 la probabilité que T donne 1 en sortie lorsqu'un élément a est tiré aléatoirement dans I_n et que l'on prend $A = f_n(a)$.

La différence entre les cas uniforme et non uniforme est donc de savoir si l'on considère un même algorithme de calcul T prenant un entier n en entrée, ou si l'on considère une suite d'algorithmes de calcul.

Cette définition apparaît à première vue assez complexe, et ceci explique qu'elle n'ait été formulée qu'assez tard. Pour l'illustrer, présentons maintenant un exemple.

Remarque 1 : On a supposé que le programme informatique a été écrit en C ou en Pascal par exemple. En fait, on peut montrer que le choix du langage informatique ne change rien : tous les langages qui ont assez d'instructions engendrent exactement les mêmes programmes. L'exécution d'un programme A dans un langage sera éventuellement plus rapide que le programme A' analogue d'un autre langage, mais le rapport des deux temps de calcul restera toujours borné par $P(n)$, où P est un polynôme fixe ne dépendant que des deux langages, et n le nombre d'instructions de A.

Remarque 2: Dans les définitions 1 et 2, on peut remarquer que "être distinguable" a été modélisé par "il existe un algorithme de complexité polynomiale qui distingue". Pourquoi modéliser par la complexité polynomiale ce que l'on peut faire, et par la complexité non polynomiale ce que l'on ne peut pas faire ("polynomial" signifie "polynomial en la taille des entrées") ? Cela provient des raisons suivantes :

1. Les fonctions qui croissent vers l'infini asymptotiquement plus vite que tous les polynômes croissent si vite que l'on peut en pratique espérer pouvoir choisir des valeurs raisonnables pous s'assurer de la sécurité. De plus, si la puissance de calcul augmente, il suffit d'augmenter légèrement ces valeurs pour rester à l'abri des attaques.

2. Cette propriété d'avoir une complexité "polynomiale" est stable par composition de fonctions.

3. On connaît des candidats qui semblent avoir les propriétés requises par ces définitions (comme ceux de la fin du paragraphe 2).

Notons cependant que l'on n'a pas de preuve absolue qu'il existe des générateurs de bits vérifiant les définitions 1 ou 2. Une telle preuve fournirait en particulier une preuve de $P \neq NP$, qui est un problème ouvert célèbre en théorie de la complexité. (Personne ne doute vraiment que $P \neq NP$, mais on ne sait pas le prouver. Si, à la surprise générale, quelqu'un prouvait que $P = NP$, il faudrait alors changer les définitions 1 et 2, et renoncer à modéliser "facile" par "polynomial" et "difficile" par "non polynomial". Nous renvoyons à [5] le lecteur intéressé par ces questions).

Exemple :
Supposons $\ell(n) = n+1$: on part donc de chaînes de n bits, et on les transforme en chaînes de $n+1$ bits.
Si $C = (c_1, ..., c_n) \in I_n$, supposons que $f_n(C) = (c_1, c_2, ..., c_n, c_1 + c_2 + ... + c_n \bmod 2)$. Alors f_n n'est pas un générateur pseudo-aléatoire de bits : en effet, à partir de $f_n(C)$, on retrouve facilement C, et de là il est facile de tester si le dernier bit vaut $c_1 + c_2 + ... + c_n \bmod 2$ ou non.
Plus précisément, considérons le programme T qui, lorsqu'il reçoit une chaîne $(\lambda_1, ..., \lambda_{n+1})$ de $n+1$ bits, regarde si $\lambda_1 + ... + \lambda_n \equiv \lambda_{n+1} \bmod 2$. Si oui, alors T donne la sortie 1. Sinon, T donne la sortie 0. Calculons P_1 et P_1^*.
P_1^* est la probabilité que T donne 1 lorsque λ_1, ..., λ_{n+1} sont tirés aléatoirement dans $\{0,1\}^{n+1}$: donc $P_1^* = \frac{1}{2}$.
Quant à P_1, c'est la probabilité que T donne 1 lorsque $(\lambda_1, ..., \lambda_{n+1}) = f_n(C)$, où C a été tiré aléatoirement dans I_n. Donc $P_1 = 1$.
Ainsi $|P_1 - P_1^*| = \frac{1}{2}$, et ceci ne décroît pas vers 0 plus vite que tout polynôme en n lorsque n tend vers l'infini. Donc T est un distingueur, et f_n n'est pas un générateur pseudo-aléatoire de bits.

En revanche, pour les exemples donnés à la fin du paragraphe 2, on ne connaît pas de distingueur.

5 Preuves relatives de sécurité

Il existe des générateurs de bits dont le caractère pseudo-aléatoire peut être prouvé à condition d'admettre certaines hypothèses de nature arithmétique. C'est en particulier le cas des générateurs RSA, BBS, quadratique centré, et de celui de Blum & Micali, que nous avons décrits au paragraphe 2. La sécurité de chacun d'entre eux repose sur une conjecture différente.
Nous allons examiner de façon plus détaillée le cas du générateur BBS, et le problème de résiduosité quadratique sur lequel il repose. Nous évoquerons ensuite plus rapidement les problèmes arithmétiques qui entrent en jeu dans la sécurité des trois autres candidats de générateurs pseudo-aléatoires évoqués ci-dessus.

5.1 Le problème de la résiduosité quadratique

Rappelons d'abord la définition des symboles de Legendre et de Jacobi.

Définition 1 : Si p est un nombre premier impair, pour tout entier $a \in \mathbf{N}$, on définit le symbole de Legendre $(\frac{a}{p})$ par :

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{si } a \equiv 0 \bmod p, \\ 1 & \text{si } a \text{ est un résidu quadratique,} \\ -1 & \text{sinon.} \end{cases}$$

Le symbole de Legendre peut se calculer de manière efficace grâce au résultat suivant :

Theorème 5.1 *Si p est un nombre premier impair, on a :*

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \bmod p.$$

Définition 2 : Soit n est un entier impair dont la décomposition en facteurs premiers s'écrit $n = p_1^{\alpha_1}...p_k^{\alpha_k}$. Pour tout entier $a \in \mathbf{N}$, on définit le symbole de Jacobi $(\frac{a}{n})$ par :

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{\alpha_i}.$$

Il peut sembler à première vue que la connaissance de la factorisation de n est nécessaire pour calculer $(\frac{a}{n})$. Heureusement, on peut s'en passer en utilisant les propriétés arithmétiques suivantes :

Propriété 1 *Si n est un entier impair, et $m_1 \equiv m_2 \bmod n$, alors :*

$$\left(\frac{m_1}{n}\right) = \left(\frac{m_2}{n}\right).$$

Propriété 2 *Si n est un entier impair, alors :*

$$\left(\frac{2}{n}\right) = \begin{cases} 1 & si\ n \equiv \pm 1 \bmod 8, \\ -1 & si\ n \equiv \pm 3 \bmod 8. \end{cases}$$

Propriété 3 *Si n est un nombre premier impair, alors :*

$$\left(\frac{m_1 m_2}{n}\right) = \left(\frac{m_1}{n}\right)\left(\frac{m_2}{n}\right).$$

Propriété 4 (Loi de réciprocité quadratique) *Si m et n sont des entiers impairs, on a :*

$$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right) & si\ m \equiv n \equiv 3 \bmod 4, \\ \left(\frac{n}{m}\right) & sinon. \end{cases}$$

Il est facile de se convaincre que ces quatre propriétés permettent de calculer le symbole de Jacobi en temps polynomial.
Dans la suite, on considère deux nombres premiers impairs p, q, et leur produit $N = pq$. La condition $(\frac{x}{N}) = 1$ est nécessaire pour qu'un entier x soit un résidu quadratique modulo N. Mais ce n'est pas une condition suffisante car $(\frac{x}{N}) = 1$ peut correspondre :

- soit à $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = 1$: on a alors bien un résidu quadratique modulo N.
- soit à $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1$: on dit alors que x est un *pseudo-carré* modulo N.

Le problème de la *résiduosité quadratique* s'énonce alors de la manière suivante : un nombre entier N produit de deux nombres premiers impairs p et q, et un entier x tel que $(\frac{x}{N}) = 1$ étant donnés, déterminer si x est ou non un résidu quadratique modulo N.
Ce problème ne peut pas être plus difficile que celui de la factorisation de N, car si on connaît p et q, on peut facilement obtenir la valeur de $(\frac{x}{p})$ et conclure. Par ailleurs, on ne connaît aucun moyen efficace de résoudre le problème de la résiduosité quadratique sans passer par la factorisation de N, et on conjecture qu'il n'existe aucun algorithme polynomial (même probabiliste) pour le résoudre.

5.2 Le générateur BBS

Comme nous l'avons vu au paragraphe 2, il est décrit pas les paramètres suivants : deux nombres premiers p et q tels que $p \equiv q \equiv 3 \bmod 4$, et leur produit $N = pq$. À partir d'une graine $x_0 \in Q(N)$, il engendre une suite (z_n) de bits définie par les formules suivantes (où $0 \leq x_n \leq N-1$) :

$$\forall n \in \mathbf{N},\ \begin{cases} x_{n+1} \equiv (x_n)^2 \bmod N \\ z_{n+1} \equiv x_{n+1} \bmod 2 \end{cases}$$

Donnons maintenant une idée des arguments permettant de ramener la sécurité du générateur BBS à la difficulté du problème de résiduosité quadratique. Pour simplifier, on se placera dans le cas uniforme.

- Supposons qu'il existe un algorithme T qui permette de distinguer les ℓ bits de sortie du générateur BBS d'une suite réellement aléatoire de I_ℓ. Plus précisément, T sort comme valeur 0 ou 1, on note P_1^* la probabilité que T donne 1 en sortie lorsque la suite est réellement aléatoire, et on note P_1 la probabilité que T prenne la valeur 1 lorsque l'on prend une graine aléatoire et que l'on envoie à T la suite alors produite par le générateur. On suppose que : $|P_1 - P_1^*| \geq \varepsilon$.

- On commence par construire une fonction B "prédicteur de bit précédent", qui – à partir des ℓ valeurs de sortie du générateur BBS – tente de "prédire" la valeur $z_0 = x_0 \bmod 2$.

  Pour cela, l'idée est d'appliquer l'algorithme T aux deux suites $(0, z_1, ..., z_{\ell-1})$ et $(1, z_1, ..., z_{\ell-1})$. L'une des deux a plus de chances de provenir d'un générateur BBS, et suivant la valeur (0 ou 1) donnée par T, et les valeurs de P_1^* et P_1, on en déduit la valeur "probable" de z_0. Plus précisément, on peut montrer que la prédiction obtenue de cette manière est juste avec une probabilité $\geq \frac{1}{2} + \frac{\varepsilon}{\ell}$ (ce résultat est un cas particulier d'un théorème général démontré par Yao en 1982, cf [13]).

- On utilise ensuite ce prédicteur pour construire un algorithme A qui distingue les résidus quadratiques modulo N des pseudo-carrés, avec une probabilité de succès supérieure à $\frac{1}{2} + \frac{\varepsilon}{\ell}$, de la manière suivante.

  Étant donné un entier x tel que $(\frac{x}{N}) = 1$, on prend $x_0 = x$ et on engendre z_1, ..., z_ℓ au moyen du générateur BBS. Puis on utilise le prédicteur de bit précédent B, qui donne une valeur z_0. Si $z_0 = x_0 \bmod 2$, alors A répond "$x \in Q(N)$", sinon A répond "$x \notin Q(N)$".

  Propriété 5 *A fournit une réponse correcte si et seulement si B prédit correctement z_0.*

  Preuve: Comme $N = pq$, avec $p \equiv q \equiv 3 \bmod 4$, on a $-1 \notin Q(N)$. Donc parmi les deux racines carrées possibles de z_1 modulo N, l'une est un résidu quadratique, et l'autre un pseudo-carré modulo N. Comme $(-z_0 \bmod N) \bmod 2 \neq (z_0 \bmod N) \bmod 2$, on voit que la réponse donnée par B permet bien à A de conclure correctement.

- L'étape suivante consiste à construire un algorithme de Monte-Carlo pour le problème de la résiduosité quadratique.

  On part d'un entier x tel que $(\frac{x}{N}) = 1$, et on cherche à savoir s'il est ou non un résidu quadratique modulo N. Pour cela on effectue les opérations suivantes :

  1. On tire r aléatoirement dans l'ensemble $(\mathbf{Z}/N\mathbf{Z})^*$ des éléments inversibles modulo N.
  2. Avec une probabilité $\frac{1}{2}$, on pose $x' = r^2 x \bmod N$ ou $x' = -r^2 x \bmod N$.
  3. On utilise l'algorithme A avec comme entrée x', ce qui donne la réponse "$x' \in Q(N)$" ou la réponse "$x' \notin Q(N)$".
  4. Si la réponse est "$x' \in Q(N)$" avec $x' \equiv r^2 x \bmod N$, ou bien si la réponse est "$x' \notin Q(N)$" avec $x' \equiv -r^2 x \bmod N$, on sort : "$x \in Q(N)$". Dans les autres cas, on sort : "$x \notin Q(N)$".

  On répète ces opérations $2m + 1$ fois. On obtient ainsi $2m + 1$ réponses, et on prend celle qui apparaît le plus souvent.

  Propriété 6 *La probabilité d'erreur de l'algorithme ainsi défini est inférieure à :*

  $$\frac{(1 - 4(\frac{\varepsilon}{\ell})^2)^m}{2}.$$

  Supposons que l'on souhaite obtenir une probabilité d'erreur inférieure à $\delta > 0$. Il suffit de choisir m tel que :

  $$\frac{(1 - 4(\frac{\varepsilon}{\ell})^2)^m}{2} \leq \delta.$$

  On peut donc prendre :

  $$m = \left\lceil \frac{1 + \log_2 \delta}{\log_2(1 - 4(\frac{\varepsilon}{\ell})^2)} \right\rceil.$$

  Il est facile de voir que ce nombre est polynomial en $\frac{1}{\delta}$ et en $\frac{\ell}{\varepsilon}$.

- Comme on conjecture qu'il n'existe pas d'algorithme de Monte-Carlo polynomial pour résoudre le problème de la résiduosité quadratique, on en déduit – sous cette hypothèse – que le le générateur BBS est bien pseudo-aléatoire.

5.3 Trois autres candidats

Il existe d'autres problèmes de théorie des nombres sur lesquels on peut s'appuyer pour établir des preuves relatives de sécurité pour des générateurs de bits. Énonçons en trois parmi les plus célèbres : le problème de la racine e-ième modulo N, le problème de la factorisation, et le problème du logarithme discret.

Problème de la racine e-ième modulo N :
On considère un entier $N = pq$, où p et q sont deux nombres premiers impairs, et un entier e premier avec $\varphi(N)$. Si y est un élément inversible modulo N, le problème consiste à trouver un entier x tel que $x^e \equiv y \bmod N$.

Problème de la factorisation : Étant donné un entier $N = pq$, où p et q sont deux nombres premiers impairs, il s'agit de retrouver les facteurs p et q à partir de N.

Problème du logarithme discret : On suppose donnés un nombre premier $p \geq 3$ et un générateur g de $(\mathbf{Z}/p\mathbf{Z})^*$. Si y est un élément non nul modulo p, il s'agit de trouver x tel que $g^x \equiv y \bmod p$.

Si l'on connaît un moyen efficace de factoriser N, le problème de la racine e-ième devient facile à résoudre. En l'état actuel des connaissance, résoudre ce problème semble aussi difficile que factoriser N (comme pour le problème de résiduosité quadratique). Comme on ne connaît aucun algorithme polynomial pour le problème de la factorisation, on dispose ainsi de deux problèmes considérés comme "difficiles" (*i.e.* non résolubles en temps polynomial). Le problème du logarithme discret est également considéré comme un problème arithmétique "difficile".
Tout comme pour le générateur BBS, qui est prouvé pseudo-aléatoire si le problème de résiduosité quadratique est difficile, on peut montrer les résultats suivants :

1. Le générateur RSA est pseudo-aléatoire si le problème de la racine e-ième modulo N est difficile.
2. Le générateur quadratique centré est pseudo-aléatoire si le problème de la factorisation est difficile.
3. Le générateur de Blum & Micali est pseudo-aléatoire si le problème du logarithme discret est difficile.

Remarque : Trouver un algorithme polynomial qui résolve le problème de la racine e-ième aurait non seulement pour conséquence de rendre inutilisable le générateur RSA, mais aussi le célèbre algorithme RSA de chiffrement à clé publique. En effet, pour chiffrer un message représenté par un entier $x \in [0, N-1]$, l'algorithme RSA, calcule $y = x^e \bmod N$. Dans ces conditions, retrouver un message à partir de son chiffré revient exactement à calculer une racine e-ième modulo N. Le fait que, depuis son invention en 1977, la communauté des mathématiciens n'ait pas trouvé de méthode pour "casser" cet algorithme, est un des arguments en faveur de la difficulté du problème de la racine e-ième.

6 Conclusion

Comme on l'a vu tout au long de cet article, le problème de la génération d'aléas sur ordinateur est assez délicat, que ce soit sur le plan de la définition conceptuelle rigoureuse, ou sur le plan de la réalisation pratique.

Les divers algorithmes proposés peuvent se classer en trois grandes catégories : ceux qui – comme le générateur congruentiel linéaire ou le générateur $1/p$ – ne vérifient par les critères de sécurité et que l'on ne peut donc pas retenir pour des applications cryptographiques par exemple, ceux dont on ne sait pas s'ils sont ou non pseudo-aléatoires (comme le générateur réducteur, qui présente l'avantage de la

rapidité, ou les générateurs à base de Triple-DES ou d'AES), et enfin ceux dont on peut affirmer qu'ils sont pseudo-aléatoires sous certaines hypothèses de nature arithmétique.

On ne connaît pour l'instant aucun générateur dont le caractère pseudo-aléatoire soit prouvé de manière inconditionnelle. L'existence de tels objets "mythiques" est liée à des problèmes profonds de théorie de la complexité, comme le célèbre "$P \neq NP$", qui reste ouvert depuis près de 25 ans...

Pour en savoir plus ...

[1] D. Coppersmith, H. Krawczyk, Y. Mansour, *The Shrinking Generator*, Advances in Cryptology, Proceedings of CRYPTO'93, Springer-Verlag, pp. 22-39.

[2] J. Daemen, V. Rijmen, *The Rijndael Page*, http://www.esat.kuleuven.ac.be/ rijmen/rijndael/

[3] M. Fürer, *The power of randomness for communication complexity*, Proc. 19th ACM Symp. on Theory of Computing, ACM, pp. 178-183.

[4] M. Furst, R. Lipton, L. Stockmeyer, *Pseudorandom number generation and space complexity*, Information and Control **64**, pp. 43-51.

[5] M. Garey, D. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979.

[6] D. Knuth, *The Art of Computer Programing*, volume 2, second edition, 1981, Addison Wesley. Voir en particulier le chapitre 3 : "Random Numbers".

[7] E. Kranakis, *Primality and Cryptography*, John Wiley and Sons, 1986. Voir en particulier le chapitre 4 : "Pseudorandom Generators".

[8] J. Lagarias, *Pseudorandom Number Generators in Cryptography and Number Theory*, Proceedings of Symposia in Applied Mathematics, volume 42, 1990.

[9] M. Luby, C. Rackoff, *How to construct pseudorandom permutations from pseudorandom functions*, SIAM J. Computing **17**, 1988, pp. 373-386.

[10] R. Rueppel, *Stream Ciphers*, in Gustavos J. Simmons, editor, *Contemporary Cryptology, The Science of Information*, IEEE Press, 1992, pp. 65-134.

[11] B. Schneier, *Applied Cryptography*, second edition, John Wiley and Sons, 1996. Voir en particulier le chapitre 16 : "Pseudo-random-sequence generators and stream ciphers".

[12] D. Stinson, *Cryptographie - Théorie et Pratique*, International Thomson Publishing France, 1996. Voir en particulier le chapitre 12 : "Générateurs pseudo-aléatoires".

[13] A. Yao, *Theory and applications of trapdoor functions (extended abstract)*, Proc. 23rd Annual Symp. on Foundations of Computer Science, IEEE, 1982, pp. 80-91.