0.2 Générateurs Linéaires Congruentiels Tronqués

0.2.1 Description

Au début des années 1980, Don Knuth (né en 1938) a suggéré la technique suivante pour produire des générateurs pseudo-aléatoires de qualité cryptographique [8]. On considère la séquence :

$$X_{i+1} \leftarrow (aX_i + b) \bmod m \qquad \text{et} \qquad Y_i \leftarrow \lfloor X_i/k \rfloor$$

On suppose que a, b, k et m sont connus. La graine du PRNG est X_0, et la séquence de ses sorties est $Y_0, Y_1, Y_2, \ldots$

C'est un générateur congruentiel linéaire *tronqué* : lors de chaque sortie, une partie de X_i nous est dissimulée, puisqu'on ne récupère que le quotient de la division euclidienne de X_i par k. Si $k = 2^\ell$, cela revient à cacher les ℓ bits de poids faible. Écrivons donc $X_i = kY_i + Z_i$ (où $0 \leq Z_i < k$ est la partie tronquée).

0.2.2 Instances

Plusieurs PRNG très connus utilisent ce principe, notamment dans la librairie C standard (les fonctions `rand()` et `mrand48()`). La norme POSIX spécifie que `mrand48` doit être implanté de la façon suivante :

```
uint64_t rand48_state;

void srand48(uint32_t seed) {
  rand48_state = seed;
  rand48_state = 0x330e + (rand48_state << 16);
}

uint32_t mrand48() {
  rand48_state = (0x00000005deece66d * rand48_state + 11) & 0x0000ffffffffffff;
  return (rand48_state >> 16);
}
```

Avec nos notations ci-dessus, cela revient à :

$$a = 25214903917, \quad b = 11, \quad m = 2^{48}, \quad k = 2^{16}$$

L'avantage, c'est que `mrand48` renvoie des entiers 32 bits. La spécification du langage C prévoit qu'une fonction `rand` soit également disponible. L'implantation suivante est suggérée dans la spécification

```
static unsigned long int next = 1;

int rand(void) { /* RAND_MAX assumed to be 32767 */
    next = next * 1103515245 + 12345;
    return ((unsigned)(next/65536) % 32768);
}

void srand(unsigned int seed) {
    next = seed;
}
```

Avec nos notations ci-dessus, cela revient à :

$$a = 1103515245, \quad b = 12345, \quad m = 2^{31}, \quad k = 2^{16}$$

Ces deux générateurs pseudo-aléatoires sont faibles, et il est possible de récupérer la graine en observant très peu de flux pseudo-aléatoire. Ceci permet de prédire la suite du flux pseudo-aléatoire, et même de connaître le flux pseudo-aléatoire généré avant l'observation.

0.2.3 Attaque lorsque $b = 0$

Le raisonnement ci-dessous est une version simplifiée de l'analyse due à Frieze, Håstad, Kannan, Lagarias et Shamir en 1988 [4].

Tout d'abord, on peut faire un raisonnement simple à base de « théorie de l'information » : on doit récupérer un état interne qui fait $\log_2 m$ bits, or chaque sortie ne contient que $\log_2 m/k$ bits d'information. On ne pourra

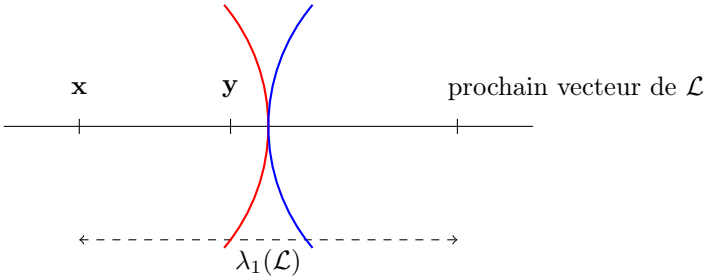

FIGURE 1 – Comment s'assurer que $\mathbf{x}$ est bien le plus proche vecteur de $\mathbf{y}$.

donc pas s'en tirer en examinant moins de $\log_2 m/(\log_2 m - \log_2 k)$ sorties. On constate que ceci est une fonction croissante de m et de k.

Notre objectif consiste à récupérer l'un des X_i, donc en fait l'un des Z_i. On ne connaît pas entièrement X_i, mais on en connaît une *approximation*, qui est $k \times Y_i$. En effet, $|X_i - kY_i| < k$, donc ces deux nombres sont « proches ».

Pour exploiter cette observation, le plan de bataille est le suivant :

- Fabriquer un réseau euclidien $\mathcal{L}$ qui contient le vecteur $\mathbf{x} = (X_0, X_1, X_2, \ldots, X_n)$ — qu'on ne connaît pas.
- Remarquer que le vecteur $\mathbf{y} = k \cdot (Y_0, Y_1, \ldots, Y_n)$ —qu'on connaît— est *proche* de $\mathbf{x}$.
- Faire le pari que $\mathbf{x}$ est le point du réseau le plus proche de $\mathbf{y}$, donc qu'on pourrait récupérer les bits manquants en résolvant une instance de CVP dans $\mathcal{L}$.

Supposons dans un premier temps que $b = 0$. On a alors $X_{i+1} \equiv aX_i \bmod m$, donc $X_i \equiv a^i X_0 \bmod m$. Comme les X_i sont des restes modulo m, il existe donc des entiers $q_1, q_2, q_3, \ldots$ (qu'on ne connaît pas) tels que :

$$\begin{aligned} X_1 &= aX_0 + q_1 m \\ X_2 &= a^2 X_0 + q_2 m \\ X_3 &= a^3 X_0 + q_3 m \\ &\vdots \end{aligned}$$

Par conséquent, on a l'égalité :

$$(X_0, q_1, q_2, \ldots, q_{n-1}) \begin{pmatrix} 1 & a & a^2 & \ldots & a^{n-1} \\ & m & & & \\ & & m & & \\ & & & \ddots & \\ & & & & m \end{pmatrix} = (X_0, X_1, X_2, \ldots, X_{n-1})$$

L'idée générale consiste à regarder le réseau $\mathcal{L}$ engendré par cette matrice. On sait que $\mathbf{x} = (X_0, X_1, X_2, \ldots, X_{n-1})$ appartient à $\mathcal{L}$ (ceci accomplit la première étape du plan de bataille).

La distance entre $\mathbf{y} = k(Y_0, Y_1, \ldots, Y_{n-1})$ et $\mathbf{x}$ est majorée par $\sqrt{k^2 + \cdots + k^2} = k\sqrt{n}$.

Le volume du réseau est m^{n-1} (le déterminant de la matrice est très facile à calculer car elle est triangulaire supérieure : c'est juste le produit des entrées sur la diagonale). On « prédit » avec l'heuristique gaussienne que le plus court vecteur de ce réseau est de taille $\lambda_1(\mathcal{L}) \approx \sqrt{n} m^{(n-1)/n}$. Par conséquent, deux points du réseau doivent être séparés d'au moins cette distance-là.

Tant que $k\sqrt{n} < \frac{1}{2}\lambda_1(\mathcal{L})$, alors il est rigoureusement certain que $\mathbf{x}$ est bien le vecteur de $\mathcal{L}$ le plus proche de $\mathbf{y}$ (cf. figure 1). Donc, si on fait confiance à notre estimation de $\lambda_1(\mathcal{L})$ par l'heuristique gaussienne, on s'attend à ce que $CVP(\mathcal{L}, \mathbf{y})$ renvoie vraiment $\mathbf{x}$ lorsque :

$$\sqrt{n}k < \sqrt{n} m^{(n-1)/n}$$

(remarque : on a laissé tomber le facteur $\frac{1}{2}$, mais de toute façon on avait déjà largué le $1/\sqrt{2e\pi}$ de l'heuristique gaussienne...). Ceci se simplifie en

$$\frac{\log m}{\log m - \log k} < n.$$

Ceci montre que l'attaque (qui consiste à résoudre CVP) fonctionne dès qu'on a observé le nombre minimal de Y_i qui permet de reconstituer X_0 de manière unique. Il faut donc résoudre une instance de CVP. Si la dimension est petite, ce sera possible avec les algorithmes exacts.

Si on ne nous donne que le bit de poids fort et que la graine fait 128 bits, par contre, on va souffrir. Un des exercice du TD discute de cette situation

0.2.4 (*) Attaque lorsque $b \neq 0$

L'idée générale consiste à se ramener au cas précédent. Pour cela, il faut faire sauter le b. Si on ne le connaît pas, on peut poser $Y_i = X_{i+1} - X_i$, car alors on a $Y_{i+1} = aY_i \bmod m$. Mais on parvient alors à reconstituer la différence entre deux états successifs et on n'est pas complètement tiré d'affaire.

Par contre, si on connaît b, la situation est plus simple. En déroulant la récurrence, on trouve :

$$\begin{aligned} X_1 &\equiv aX_0 + b \mod m \\ X_2 &\equiv a^2X_0 + (a+1)b \\ X_3 &\equiv a^3X_0 + (a^2+a+1)b \\ &\vdots \\ X_i &\equiv a^iX_0 + b\sum_{i=0}^{i-1} a^i \end{aligned}$$

Donc on pose $X'_i = X_i - b\sum_{i=0}^{i-1} a^i$ (si on retrouve l'un des X'_i, on retrouve automatiquement le X_i correspondant). Le but de la manoeuvre, c'est qu'alors $X'_0 = X_0$ et $X'_i = a^iX'_0$.

En fait, X' est la séquence d'états internes à laquelle on aurait affaire si $b = 0$. Comme $Y_i = \lfloor X_i/k \rfloor$, il semble logique de poser $Y'_i = \lfloor X'_i/k \rfloor$. On peut calculer les Y'_i avec l'observation :

$$Y'_i = \left\lfloor \frac{X_i}{k} - \frac{b}{k}\sum_{i=0}^{i-1} a^i \right\rfloor = Y_i - \left\lfloor \frac{b}{k}\sum_{i=0}^{i-1} a^i \right\rfloor \pm 1$$

Le ± 1 vient du fait que la partie entière peut introduire une « erreur d'arrondi ». Il n'empêche pas que $\mathbf{y}' = k(Y'_0, \ldots, Y'_{n-1})$ est proche d'un vecteur du réseau $\mathcal{L}$ qui révèle les X'_i.

0.2.5 (*) Cas des paramètres inconnus

Antoine Joux (né en 1967) et Jacques Stern (né en 1949) ont montré en 1998 [6] que même si on ne connaît pas a, b et m, on peut tout retrouver la plupart du temps. Les techniques utilisées sont plus sophistiquées... mais font aussi appel aux réseaux euclidiens. En deux mots, il s'agit de produire une liste de polynômes P_i à coefficients entiers tels que $P_i(a) \equiv 0 \bmod m$. On les obtient en cherchant de « petites relations linéaires » entre les Y_i avec LLL.

Une fois qu'on les a, on peut en faire des combinaisons linéaires pour éliminer toutes les puissances de X, et on se retrouve avec un (petit) multiple de m.

0.3 Le problème du sac à dos (Subset sum)

Le problème Subset Sum est un problème NP-complet classique [4]. Étant donné k entiers $a_1, \ldots, a_k$ et un entier « cible » b, il s'agit de décider s'il existe des coefficients $x_i \in \{0, 1\}$ tels que $b = \sum x_i a_i$ — autrement dit, la cible b est-elle la somme d'un sous-ensemble des a_i.

On peut aussi, de manière plus commode, s'intéresser à une version modulaire du problème : existe des coefficients $x_i \in \{0, 1\}$ tels que $b = \sum x_i a_i \bmod M$.

La version décisionnelle (qui est NP-dure, c'est l'un des 21 problèmes NP-Complet de Karp [7]) consiste à déterminer si les x_i existent ; la version calculatoire consiste à les trouver. Aucun algorithme capable de s'exécuter en temps polynomial dans le pire des cas sur des ordinateurs classiques ou quantiques n'est connu à ce jour. Par conséquent, il est tentant de faire de la cryptographie en se basant sur la difficulté supposée de ce problème.

4. Dans le contexte de la cryptographie, on l'appelle parfois le problème du sac-à-dos (Knapsack).