Rapport sur le schéma 1

AJUELOS Emmanuel - LAY Kenny - NAMET Michael

Nous nous sommes vus attribuer le schéma suivant :

> ▷ **Schéma 1 :** On considère le générateur pseudo-aléatoire défini par :
>
> $$X_{i+1} \leftarrow (2^{61} - 1) \cdot X_i \bmod (2^{127} - 1) \qquad \text{et} \qquad Y_i \leftarrow X_i \bmod 2^8$$
>
> La graine du PRNG est X_0 (elle fait 127 bits), et la séquence de ses sorties est $Y_1, Y_2, \ldots$

Vous trouverez tout notre code et documentation sur le répertoire GitHub : https://github.com/ajuelosemmanuel/CRYPTA_Project

Implantation du schéma

Nous avons commencé par implanter ce schéma naïvement en Python. Une seconde version est implantée dans le but d'avoir une trace des différents *X* et *Y*.

Nous nous apercevons que le schéma correspond à une suite dont la formule générale est :

$$X_n = X_0 * (2^{61} - 1)^n \bmod (2^{127} - 1)$$

Une seconde implantation est mise en place utilisant cette formule.

Statistiques et efficacité du schéma

Afin d'évaluer l'efficacité du schéma, nous suivons certaines méthodes conseillées par le NIST qui sont normalement applicables pour les générateurs de bits aléatoires. Nous les avons "adaptées" pour la génération de nombres plus grands en concaténant les bits de chaque sortie. Par exemple, si nous n'avions généré que trois sorties, par exemple 65, 241 et 137, la chaîne aurait été "010000011111000110001001".

Pour les statistiques, nous avons généré 10000 nombres pour 1000 graines et *a* aléatoires, donc 10 000 000 nombres. Le code se trouve ici.

La première méthode est de compter le nombre de “0” et de “1” générés, et de vérifier si on se rapproche de 50%. Ici, nous obtenons 49.99715625% de 0, ce qui est très proche de 50%. Les bits “seuls” ont l’air d’être générés aléatoirement.

La seconde méthode est l’utilisation de la transformation de Fourier discrète, pour faire de l’analyse spectrale. L’objectif de cette méthode est de retrouver des motifs dans les bits générés, comme expliqué dans cette fonction. Ici, notre *p-value* vaut 0.8736478816189553, et est donc proche de 1., ce qui indique que notre sortie est aléatoire (si la *p-value* valait 1, la sortie serait parfaitement aléatoire).

Le générateur semble donc, que ce soit en regardant les bits dans leur ensemble ou individuellement, totalement aléatoire. Le schéma semble donc efficace.

Implantation de l’attaque

Pour implanter cette attaque, nous nous sommes appuyés sur la partie du cours sur l’attaque d’un générateur linéaire congruentiels tronqués dans le cas où $b = 0$.

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

Cependant, ce sont les bits de poids forts qui sont donnés dans le cours. Ici, ce sont les bits de poids faible qui nous sont donnés. Après avoir lu “Reconstructing truncated integer variables satisfying linear congruences”, et tout particulièrement la partie 2.7, nous commençons par inverser 2^8 mod (2^{127} - 1). L’inverse de 2^8 mod (2^{127} - 1) existe puisque 2^{127} - 1 est premier et impair. Nous le notons I. Cela nous permettra de récupérer les bits de poids forts manquants. En effet, pour chaque X_i donné, nous avons :

$$X_i = 2^8 * Z_i + Y_i \bmod (2^{127} - 1)$$

avec Y_i les 8 derniers bits donnés et Z_i les bits de poids forts que nous ne connaissons pas.

En multipliant les deux côtés par I, nous nous retrouvons avec :

$$X_i * I = Z_i + Y_i * I \bmod (2^{127} - 1)$$

Ainsi, nous avons $Y_i * I > Z_i$, Z_i étant un nombre de 119 bits et ($Y_i * I$) un nombre de 127 bits (qui est donc 256 fois plus grand). D'après le cours, avec suffisamment d'exécutions successives, il est possible de retrouver les ($X_i * I$).

Nous avons donc réalisé plusieurs tests et avons pu déterminer ceci : pour 10000 essais aléatoires et avec 19 Y_i consécutifs, nous avons 86,4% de réussite pour retrouver les X_i correspondants (le script de génération des statistiques pour cette attaque se trouve ici). Nous passons à 100% de réussite avec 21 Y_i consécutifs. Il est possible de démontrer pourquoi ce nombre d'après "Practical seed-recovery for the PCG Pseudo-Random Number Generator" partie 3.2, mais nous n'avons pas eu le temps de le faire.

Pour revenir à l'implantation de l'attaque, les étapes sont donc les suivantes :

- Inverser $2^8 \bmod (2^{127} - 1)$
- Générer le réseau euclidien L basé sur les valeurs $a = 2^{61} - 1$, $p = 2^{127} - 1$ et $n = 21$
- Résoudre une instance du CVP à partir de L et du vecteur composé des n premières sorties successives multipliées par I
- Le vecteur résultant correspond aux 21 premiers ($X_i * I$) : il est maintenant possible de retrouver tous les X_i avec $X_i = X_i * I * 2^8 \bmod (2^{127} - 1)$
- Il est alors possible de retrouver les prochains Y_i en multipliant le dernier X_i par a et en gardant ses 8 derniers bits, et ainsi de suite

Nous tentons maintenant de retrouver la graine X_0. Tout d'abord, nous inversons $a = 2^{61} - 1 \bmod (2^{127} - 1)$. Puis, nous supposons que X_0 est égal au produit du premier X_i (calculé à partir du vecteur résultant du CVP) et de l'inverse de a, le tout modulo $(2^{127} - 1)$. Soit :

$$X_0 = Xi[0] * a^{-1} \bmod (2^{127} - 1), \text{ avec } Xi \text{ le vecteur résultant du CVP}$$

Suite à cela, nous calculons les Y_i' correspondants. Si pour tout i, $Y_i' = Y_i$, alors nous avons retrouvé la graine.

Challenge

Après avoir implanté l'attaque pour le cas général en Python, nous avons créé une version dédiée au challenge. Nous utilisons quelques fonctions annexes et la librairie fpylll, afin de calculer des inverses modulaires et de résoudre une instance de CVP. Le code correspond à une traduction en Python du raisonnement décrit plus haut.

Pour vérifier que la graine obtenue est bien la bonne, nous générons une liste des 65 premiers Y à partir de notre graine trouvée. Si cette liste est égale à celle fournie dans l'énoncé du challenge, alors nous avons réussi à le casser et à obtenir la graine.

La vérification a bien fonctionné dans notre cas, et nous avons trouvé :

X_0 = 160132562724753331766255120961262537267

Contremesure

Afin de contrer l'attaque précédemment implantée, il faut empêcher l'attaquant d'avoir les bits de poids fort. Une solution simple est de rendre impossible l'inversion de 2^8, ce qui est facile : il suffit de remplacer le modulo utilisé (2^{127} - 1) par un nombre pair. En effet, un nombre pair n'a pas d'inverse modulo [un autre nombre pair].

Comment la sécurité serait-elle affectée si certains paramètres étaient agrandis/rétrécis

D'après "Reconstructing truncated integer variables satisfying linear congruences" partie 3.2, il suffit de 63 sorties consécutives pour retrouver une graine ayant la même taille en n'ayant que les 6 derniers bits.

De manière générale, réduire le nombre de bits renvoyés ne change que le nombre de sorties nécessaires pour retrouver la graine, mais ne nous empêche pas de la retrouver.
Rendre les états internes, la graine et *a* plus grands reviendrait à la même chose, car ce schéma est "fondamentalement" faible.

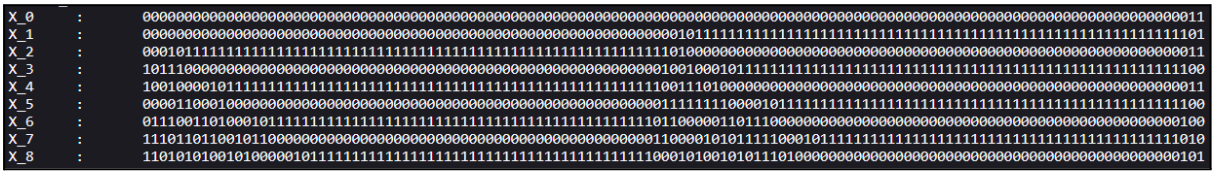
Cependant, en ne renvoyant qu'un seul bit (ou peu de bits, mais on faciliterait la tâche de l'attaquant) et en ayant des états internes très grands, il serait très difficile (voire impossible) de retrouver la graine avec les ordinateurs actuels - mais ce ne serait pas "théoriquement" sûr.

Évidemment, réduire la taille des états internes, de la graine et de *a* et renvoyer plus de bits ("faire grandir *Y*") rendrait le schéma beaucoup plus vulnérable, car il faudrait encore moins de sorties successives pour l'attaquer.

Notes et remarques

Dans un premier temps, le schéma donné n'était qu'un cas particulier, avec $a = 2^{61} - 1$. Nous avions alors quelques observations :

- Avec des graines de petite taille, on voit un certain motif. Exemple avec $X_0 = 3$:



- Avec plus de calculs, nous observons l'égalité suivante : $X_i + X_{i+1} = X_i * 2^{61} \bmod (2^{127} - 1)$ et donc : $X_{i+1} = X_i << 61 - X_i \bmod (2^{127} - 1)$, ce qui explique le motif. Nous avions donc, pour toute graine inférieure $X_0 < 2^6$, $Y_2 = X_0$.

- Il était possible, avec le solveur Z3, de retrouver la graine avec 25 Y_i consécutifs