Attacks on Pseudo Random Number Generators Hiding a Linear Structure

Florette Martinez
florette.martinez@lip6.fr

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Abstract. We introduce lattice-based practical seed-recovery attacks against two efficient number-theoretic pseudo-random number generators: the *fast knapsack generator* and a family of *combined multiple recursive generators*. The fast knapsack generator was introduced in 2009 by Von Zur Gathen and Shparlinski. It generates pseudo-random numbers very efficiently with strong mathematical guarantees on their statistical properties but its resistance to cryptanalysis was left open since 2009. The given attacks are surprisingly efficient when the truncated bits do not represent a too large proportion of the internal states. Their complexities do not strongly increase with the size of parameters, only with the proportion of discarded bits.
A multiple recursive generator is a pseudo-random number generator based on a constant-recursive sequence. A combined multiple recursive generator is a pseudo-random number generator based on combining two or more multiple recursive generators. L'Écuyer presented the general construction in 1996 and a popular instantiation deemed MRG32k3a in 1999. We use algebraic relations of both pseudo-random generators with underlying algebraic generators to show that they are cryptographically insecure. We provide a theoretical analysis as well as efficient implementations.

Keywords: Pseudo-random number generators, Knapsack problem, Coppersmith Methods, Cryptanalysis.

1 Introduction

A pseudo-random number generator (PRNG) is an efficient deterministic algorithm that stretches a small random seed into a longer pseudo-random sequence of numbers. These generators can be used to emulate randomness in games, numerical simulations or cryptographic protocols. These different situations call for PRNGs with different properties. A cryptographic application will need a strong PRNG that produces a sequence of bits indistinguishable from "truly" random bits by efficient adversaries while a numerical simulation or a game will ask for a fast and light PRNG.

Analysing the quality of randomness for a PRNG suited for cryptographic applications is natural as a failure in these PRNGs would lead to problematic

security breaches. Rueppel and Massey introduced the *knapsack generator* [16] in 1985 for cryptographic purposes. One chooses n secret bits $u_0, \dots u_{n-1}$ and n secret weights $\omega_0, \dots, \omega_n$ to form the seed. A linear feedback shift register (LFSR) generates the control bits (u_i) from the n secret bits and a public feedback polynomial of order n. At step i, the generator computes $v_i \equiv \sum_{j=0}^{n-1} u_{i+j}\omega_j \mod 2^n$, discards the least significant bits and outputs the remaining. In 2009, Knellwolf and Meier [11] presented the main attack against this generator . They used a guess-and-determine strategy coupled with lattice-based techniques to recover most of the key in relevant instances of the generator. In order to run said attack, they needed to guess all the n initial control bits. Hence their attack had a time complexity $\Omega(2^n)$. This attack is not fast enough to definitively keep the knapsack generator away from cryptographic applications. In 2009, Von zur Gathen and Shparlinski presented a faster and lighter version of the knapsack generator called the *fast knapsack generator* [7]. The main modification was a specialisation of the weights. In their paper, the authors mention that it was not clear if that specialisation had an impact on the security of this generator. Thus it was not known if it was suited for cryptographic purposes. In this article, we notice similarities between the fast knapsack generator and a linear congruential generator (LCG). Because of the specialisation of the weights, the fast knapsack generator tends to act like a LCG one iteration out of four. We present here lattice-based attacks exploiting this new weakness. We first describe three different algorithms to attack the underlying LCG, two using Coppersmith Methods and one based on Stern's attack against the LCG. Then we present how such algorithms can be used to break the fast knapsack generator. These algorithms allow us to completely recover the seed when less than a quarter of the bits are discarded.

Attacking a non-cryptographic PRNG is not irrelevant. Non-cryptographic PRNGs tend to be faster and lighter than their cryptographic counterparts. As they do not pretend to achieve some kind of security, they are less studied by cryptanalysts hence there might not exist any known attack against them. Because of that, one might be tempted to replace a strong but slow cryptographic PRNG with a faster non-cryptographic one. Breaking non-cryptographic PRNGs could deter anyone to use them outside of what they are made for. This had already been done with the PCG64 by Bouillaguet et al. in 2020 [2]. The PCG64 is the default pseudo-random number generator in the popular NumPy [18] scientific computing package for Python.

A combined linear congruential generator (CLCG) is a pseudo-random number generator algorithm based on combining two or more linear-congruential generators. The general construction was proposed in 1982 by Wichmann and Hill in [19]. A multiple recursive generator (MRG) is a pseudo-random generator based on a constant-recursive sequence. Like the LCGs, the MRGs can be combined to obtain CMRGs. In 1999, L'Écuyer presented a family of parameters giving CMRGs with good properties. These PRNGs are fast and pass the "spectral test" evaluating their closeness to the uniform distribution. The more famous of these CMRGs is the MRG32k3a, largely used for producing multi-

ple streams of pseudo random numbers, as seen in [13]. It is one of the PRNGs implanted in Matlab and the native PRNG of the programming language Racket.

This PRNG had already be used once in place of a secure one for the website *Hacker news*. In 2009, Franke [4] managed to hack this website and was able to steal accounts. His attack was not based on breaking the MRG32k3a but on guessing how the seed was generated. In fact the seed was the time (in milliseconds) when the Hacker News software was last started. After crashing the website he had access to the information he needed. In this case, breaking the MRG32k3a could have lead us to an other real life attack against this website. In our paper we will present an attack against CMRGs that output the difference between two MRG of order three. The trick will be to see the two congruential constant-recursive sequences as two projections of a single larger congruential constant-recursive sequence. This attack will cover the particular case of the MRG32k3a. Even if we reduce our study to those specific CMRGs, the same techniques can be used on CMRGs combining more than two MRGs or MRGs of larger orders.

Attacking a non-cryptographic PRNG is not only security-related. As mentioned earlier, PRNGs can be used in numerical simulations and a hidden structure in a PRNG could cause bias in said simulation. In [3], Ferrenberg et al. ran classical Ferromagnetic Ising model Monte-Carlo simulations in specific cases where exact results were known, with different PRNGs. They observed that the choice of the PRNG had a significant impact on the outcome. For example, a given LFSR tented to give energy levels that were too low and a critical temperature that was to high.

In Section 2, we will present a simplified version of the Coppersmith method, used in the attacks against both PRNGs. The different attacks on the fast knapsack generator will be discussed in Sections 3 and 4 while the attack against the CMRGs will be presented in Section 5.

2 Coppersmith Method

In this section, we give a short description of a Coppersmith method used to solve a multivariate modular polynomial system of equations over a single modulus. We refer the reader to [8] for proofs.

We consider $P_1(y_0, \ldots, y_n), \ldots, P_s(y_0, \ldots, y_n)$ s irreducible multivariate polynomials defined over $\mathbb{Z}$, having a common small root $(x_0, \ldots, x_n)$ modulo a known integer N. Said root is said small because it must be bounded by known values, namely $|x_0| < X_0, \ldots, |x_n| < X_n$. In order to find this root, we may want to increase the number of polynomials by adding polynomials of the form $y_1^{k_1} \ldots y_n^{k_n} P_i^{k_{n+1}}$. We suppose we have now r polynomials $P_1, \ldots, P_r$ linearly independent but not necessarily irreducible. To each of these polynomials P_i we associate a number k_i that will be the multiplicity of $(x_0, \ldots, x_n)$ as a root of $P_i \bmod N$ (in other terms, k_i is the largest integer such that $P_i(x_0, \ldots, x_n) \equiv 0 \bmod N^{k_i}$). We construct the matrix $\mathcal{M}$ as follows:

$$\mathcal{M} = \begin{array}{c} \begin{array}{ccccc} & & & P_1 \ \cdots & P_r \\ & & & \downarrow \ \cdots & \downarrow \end{array} \\ \left(\begin{array}{ccc|ccc} 1 & & & & & \\ & X_0^{-1} & & & \star & \\ & & \ddots & & & \\ & & X_0^{-a_0} \dots X_n^{-a_n} & & & \\ \hline & & & N^{k_1} & & \\ & 0 & & & \ddots & \\ & & & & & N^{k_r} \end{array}\right) \end{array} \begin{array}{c} 1 \\ y_0 \\ \vdots \\ y_0^{a_0} \dots y_n^{a_n} \\ \\ \\ \\ \end{array}$$

We denote $\mathfrak{M}$ the set of monomials that appear at least in one P_i and $|\mathfrak{M}|$ its cardinality. Each one of the upper rows (between 1 and $|\mathfrak{M}|$) corresponds to one of these monomials and each one of the latest columns (from $|\mathfrak{M}|+1$ to $|\mathfrak{M}|+r$) corresponds to one of the polynomials.

Let i be in $\{1, \dots, |\mathfrak{M}|\}$, we denote m_i the i-th monomial of $\mathfrak{M}$, $m_i = y_0^{b_0} \dots y_n^{b_n}$. The value of $\mathcal{M}_{i,i}$ will be the inverse of the bound on m_i, hence $X_0^{-b_0} \dots X_n^{-b_n}$. For all j between 1 and r, the value of $\mathcal{M}_{i,|\mathfrak{M}|+j}$ will be the coefficient of m_i in P_j. Finally, the value of $\mathcal{M}_{|\mathfrak{M}|+j,|\mathfrak{M}|+j}$ will be k_j as described in the previous paragraph.

We want to show that the smaller vector of the lattice spanned by the rows of $\mathcal{M}$ contains the solution $(x_0, \dots, x_n)$. We denote by c_i the integer such that $P_i(x_0, \dots, x_n) = c_i N^{k_i}$. We can construct v:

$$\begin{aligned} v &= (1, x_0, \dots, x_0^{a_0} \dots x_{m-1}^{a_n}, -c_1, \dots, -c_r) \times \mathcal{M} \\ &= \left(1, \frac{x_0}{X_0}, \dots, \frac{x_0^{a_0} \dots x_n^{a_n}}{X_0^{a_0} \dots X_n^{a_n}}, 0, \dots, 0\right). \end{aligned}$$

By construction, the vector v is in the lattice. Its first $|\mathfrak{M}|$ coordinates are smaller than one and the remaining ones are null, hence it is a small vector. In general, retrieving the shortest vector of a lattice is a hard problem (called the SVP for Shortest Vector Problem), but if this short vector is abnormally short, it can be far easiest. A common method to find such a vector is applying the LLL algorithm to the lattice. The LLL [12] is a polynomial-time reduction algorithm presented by Lenstra, Lenstra and Lovász in 1982. It takes as input a basis of a lattice and outputs a short and nearly orthogonal basis of the same lattice. The smallest vector of this basis is, as a consequence, a small vector of the lattice. We will thus apply LLL on the matrix $\mathcal{M}$ to obtain the small vector v.

The conditions on the bounds that make this method works are given by the following (simplified) equation:

$$\prod_{y_0^{a_0} \dots y_n^{a_n} \in \mathfrak{M}} X_0^{a_0} \dots X_n^{a_n} < N^{\sum_{i=1}^{r} k_i}. \tag{1}$$

For further details see [15].

3 Attacks on the Linear Congruential Generator

A *Linear Congruential Generator* (LCG) is a pseudo-random number generator whose internal states are of the form $v_{i+1} = zv_i + C \bmod N$. The parameter z is called the multiplier, C is the increment and N is the modulus. Those generators have been largely studied in various cases. In 1984, Frieze et al. [6] showed that, provided both the modulus N and and the multiplier z, the sequence output by a LCG was completely predictable as long as more than $2/5$ of the bits were output. In 1987, Stern [17] presented two algorithms to predict a sequence output by a LCG with $\mathcal{O}(\sqrt{n})$ outputs. The first algorithm treated the case where only the modulus N was known and the second one treated the case where all the parameters were secret. In 1988, Frieze and al. proposed in [5] a polynomial-time algorithm to retrieve the seed when the multiplier z and the modulus N were known. In 1997, Joux and Stern presented in [9] a polynomial-time algorithm against the LCG to retrieve the parameters z, c and N when they are kept secret.

3.1 Attacks via a Coppersmith Method

In the following, we will study the LCG underlying in the fast knapsack generator. This LCG is particular in the sense that z is unknown, $c = 0$ and $N = 2^n$. We have two options to retrieve the seed of this generator. We can create new attacks specifically against this type of LCG or adapt existing attacks like Stern's. In this subsection we will explore the first option and use our own algorithm based on a Coppersmith method to retrieve z. We also notice that our strategy is easy to adapt to the case where the outputs we have are no longer consecutive.

Let v_0 and z be two n-bits integers. The integer v_0 is the *seed* and z the *multiplier*. We choose z odd (hence coprime to 2^n), otherwise v_1 would be divisible by 2, v_2 by 2^2 and v_k by 2^k. At step $i+1$, our LCG computes $v_{i+1} = z \times v_i \bmod 2^n$ and outputs the $n - \ell$ most significant bits.

For an internal state v_i, we introduce the following notations:

- $H_i = (v_i \ quo \ 2^\ell) \times 2^\ell + 2^{\ell-1}$, where *quo* denotes the quotient of the Euclidean division (H_i is constructed from the output, hence it is known)
- $\delta_i = v_i - H_i$ (δ_i represents the ℓ discarded bits, it is unknown)

Attack 1: Consecutive outputs Let v_0, v_1, v_2 be 3 consecutive internal states of the LCG. We have $v_1 = zv_0 \bmod 2^n$ and $v_2 = zv_1 \bmod 2^n$. As z and 2^n are coprime, we obtain:

$$v_1^2 = v_0 v_2 \bmod 2^n.$$

We replace v_i by $H_i + \delta_i$:

$$H_1^2 + 2H_1\delta_1 + \delta_1^2 = H_0H_2 + H_0\delta_2 + H_2\delta_0 + \delta_0\delta_2 \bmod 2^n,$$

and notice that $(\delta_0, \delta_1, \delta_2)$ is a small root of the polynomial $P \bmod 2^n$ where

$$P(y_0, y_1, y_2) = y_1^2 - y_0y_2 + 2H_1y_1 - H_0y_2 - H_2y_0 + H_1^2 - H_0H_2.$$

We will apply the Coppersmith method on P with bounds $X_0 = X_1 = X_2 = 2^\ell$. The set of monomials is $\mathfrak{M} = \{y_0, y_1, y_2, y_1^2, y_0y_2\}$ hence we should heuristically recover the root if $(2^\ell)^7 = X_0 \times X_1 \times X_2 \times X_1^2 \times X_0X_2 < 2^n$, that is to say if $\ell/n < 1/7$.

Generalization Let $v_0, \ldots, v_k$ be $k+1$ consecutive internal states. We will obtain $\binom{k}{2}$ equations of the form $v_jv_{i+1} = v_iv_{j+1} \bmod 2^n$. Hence we will construct k polynomials P_i of which $(\delta_0, \ldots, \delta_k)$ is a simple root mod 2^n. The set of appearing monomials will be:

$$\mathfrak{M} = \{y_i | i \in \{0, \ldots, k\}\} \bigcup \{y_iy_{j+1} | i, j \in \{0, \ldots, k-1\}, i \neq j\}.$$

We find that $\prod_{y_i | i \in \{0,\ldots,k\}} X_i \times \prod_{y_iy_{j+1} | i,j \in \{0,\ldots,k-1\}, i \neq j} X_iX_{j+1} = (2^\ell)^{\Gamma(k)}$ where $\Gamma(k) = (k+1) + 2 \times 2\binom{k}{2}$. Thus the attack should work as long as $\ell/n < \binom{k}{2}/\Gamma(k)$. This theoretical bound increases toward $1/4$.

As usual with Coppersmith methods, the theoretical bound is smaller than what we can really achieve.

k	2	3	4	5
ℓ/n (theoritical) $<$	1/7	3/16	6/29	5/23
ℓ/n (experimental) $<$	0.3	0.35	0.38	0.40

We also present the computing times for different n and k.

$n \backslash k$	2	3	4	5
32	$0.002s$	$0.005s$	$0.01s$	$0.04s$
64	$0.003s$	$0.009s$	$0.03s$	$0.1s$
1024	$0.02s$	$0.1s$	$0.7s$	$2s$

These computing times are averages of a hundred instances of the algorithm running on a standard laptop: a Dell Latitude 7400, running on Ubuntu 18.04 with Sagemath version 8.1. The same laptop with the same configuration will be used for the rest of the experiments of this paper.

Remark 1 *As mention in Section 2, we could try to optimise our Coppersmith method by adding polynomials of the form $y_1^{k_1} \ldots y_n^{k_n} P_i^{k_{n+1}}$, but we refrained for two main reasons. The first one is that without any other polynomial, the size of our lattice remains small and our attack practical. The second reason is that we tried in Appendix B to find a suitable family of polynomials to improve our attack and the results are not encouraging. This will remains true for the next attack*

Attack 2: Sparse outputs Now we suppose we have two pairs of two consecutive internal states (v_0, v_1) and (v_i, v_{i+1}). Then $(\delta_0, \delta_1, \delta_i, \delta_{i+1})$ is a small root of P mod 2^n where

$$P = y_0y_{i+1} - y_1y_i + H_0y_{i+1} + H_{i+1}y_0 - H_1y_i - H_iy_1 + H_0H_{i+1} - H_1H_i.$$

We will apply the Coppersmith method on P with $X_0 = X_1 = X_i = X_{i+1} = 2^\ell$. The set of monomials is $\mathfrak{M} = \{y_0, y_1, y_i, y_{i+1}, y_0 y_{i+1}, y_1 y_i\}$ hence we should heuristically recover the root if $(2^\ell)^8 = X_0 \times X_1 \times X_i \times X_{i+1} \times X_0 X_{i+1} \times X_1 X_i < 2^n$, that is to say if $\ell/n < 1/8$.

Generalisation Let S be a set of k distinct integers (the bigger being i_S) and $\bigcup_{i \in S} \{v_i, v_{i+1}\}$ be at most $2k$ internal states. We will obtain $\binom{k}{2}$ equations of the form $v_j v_{i+1} = v_i v_{j+1} \bmod 2^n$ hence $\binom{k}{2}$ polynomials P_i of which $(\delta_0, \ldots, \delta_{i_S+1})$ is a simple root mod 2^n. The set of appearing monomials will be:

$$\mathfrak{M} = \{y_i, y_{i+1} | i \in S\} \bigcup \{y_i y_{j+1} | i, j \in S, i \neq j\}.$$

We will have at most $2k$ monomials of degree 1 and $2\binom{k}{2}$ monomials of degree 2. Heuristically, our attack should work if $(2^\ell)^{2k+4\binom{k}{2}} < (2^n)^{\binom{k}{2}}$. In other words, our attack should work if $\ell/n < \frac{k-1}{4k}$. This theoretical bound increases toward $1/4$.

As usual with Coppersmith methods, the theoretical bound is smaller than what we can really achieve.

k	2	3	4	5	6	7	8
ℓ/n (theoritical) $<$	1/8	1/6	3/16	1/5	5/24	3/14	7/32
ℓ/n (experimental) $<$	0.16	0.25	0.31	0.34	0.36	0.38	0.4

We also present the computing time for different n and k.

$n \backslash k$	2	3	4	5	6	7	8
32	$0.001s$	$0.003s$	$0.008s$	$0.02s$	$0.04s$	$0.09s$	$0.2s$
64	$0.002s$	$0.004s$	$0.02s$	$0.04s$	$0.08s$	$0.2s$	$0.5s$
1024	$0.003s$	$0.2s$	$0.2s$	$0.8s$	$1.6s$	$5s$	$13s$

These computing times are averages of a hundred instances of the algorithm.

3.2 Attack 3: with Stern's algorithm

As mentioned earlier, one strategy to attack our simple LCG was to adapt existent attacks to the case where $C = 0$ and $N = 2^n$. Here we are going to describe and adapt Stern's attack against the LCG, presented in [17].

Let us consider a LCG with internal states given by $v_{i+1} = z v_i + C \bmod N$ with v_0, z, C and N secret. To obtain the output y_i, we discard the last ℓ bits of the internal state v_i.

First part of Stern's algorithm Let K, t and r be three integer parameters to be discussed later, with $r > t$. We consider the vectors:

$$Y_i = \begin{pmatrix} y_{i+1} - y_i \\ y_{i+2} - y_{i+1} \\ \vdots \\ y_{i+t} - y_{i+t-1} \end{pmatrix} \text{ and } V_i = \begin{pmatrix} v_{i+1} - v_i \\ v_{i+2} - v_{i+1} \\ \vdots \\ v_{i+t} - v_{i+t-1} \end{pmatrix}.$$

Then we construct the following matrix:

$$\left(\begin{array}{c|ccc} KY_0 & 1 & & \\ KY_1 & & 1 & \\ \vdots & & & \ddots \\ KY_{r-1} & & & 1 \end{array}\right),$$

apply LLL and obtain a small vector $(K\sum_{i=0}^{r-1}\lambda_i Y_i, \lambda_0, \ldots, \lambda_{r-1})$. We will have to choose K big enough to force the λ_i's to satisfy:

$$\sum_{i=0}^{r-1}\lambda_i Y_i = 0.$$

The key point here is that, with t and r well chosen, we can expect:

$$\sum_{i=0}^{r-1}\lambda_i V_i = 0,$$

hence if we consider the polynomial $f(X) = (v_2 - v_1)\sum_{i=1}^{r}\lambda_i X^{i-1}$, it satisfies $f(z) \equiv 0 \bmod N$. In the case where N is prime, the integers parameters K, t, r must satisfy:

$t > n/(n-\ell)$
$r \approx \sqrt{2(n-\ell)t}$
and $K = \lceil \sqrt{r}2^{(r-1)/2}B \rceil$

where $B = 2^{t((n-\ell)+\log r+1)/(r-t)}$ (see [17, 9]).

If we were to use directly this method on our particular LCG (with $C = 0$ and $N = 2^n$) we would face two major inconveniences. The first one is the number of outputs needed. Let suppose $n = 1024$ and $\ell = 128$, then $\ell/n = 1/8$ and we know our algorithm using a Coppersmith method would need 3 outputs to recover z. Here, Stern's algorithm would need a bit more than 60 LCG outputs and we cannot expect the fast knapsack generator to behave like a LCG sixty times in a row. The second inconvenience is the proof of this algorithm. The fact that N is prime (or almost prime) is crucial as it allows us to count the roots of a certain polynomial modulo N. In our case $N = 2^n$ hence we cannot bound the number of roots of a polynomial modulo 2^n any more.

Modified algorithm What we can do is trust the algorithm to works even if the proof does not hold any more and search for the right parameters experimentally. But before, we are going to do some modifications. As we know that $C = 0$, we can replace the vectors Y_i and V_i by:

$$Y_i' = \begin{pmatrix} y_i \\ y_{i+1} \\ \vdots \\ y_{i+t-1} \end{pmatrix} \text{ and } V_i' = \begin{pmatrix} v_i \\ v_{i+1} \\ \vdots \\ v_{i+t-1} \end{pmatrix}.$$

Then instead of searching a small vector $(\lambda_0, \ldots, \lambda_{r-1})$ such that:

$$\sum_{i=0}^{r-1} \lambda_i Y'_i = 0,$$

we are going to search for an even smaller vector $(\mu_0, \ldots, \mu_{r-1})$ such that:

$$\sum_{i=0}^{r-1} \mu_i Y'_i = 0 \bmod N$$

(in fact we do not even need this sum to be zero, we just need it to be small). To find such a vector we construct the following matrix:

$$\left(\begin{array}{ccc|ccc} KY'_0 & & & 1 & & \\ KY'_1 & & & & 1 & \\ \vdots & & & & & \ddots \\ KY_{r-1'} & & & & & 1 \\ \hline K2^n & & & & & \\ & \ddots & & & 0 & \\ & & K2^n & & & \end{array}\right)$$

and apply LLL.

Then if the vector satisfies $\sum_{i=0}^{r-1} \mu_i V'_i = 0 \bmod N$ and v_0 is odd, the polynomial $f(X) = \sum_{i=0}^{r-1} \mu_i X^i$ will satisfies $f(z) \equiv 0 \bmod N$. When we have this polynomial, we compute its roots modulo 2^n with a lift of Hensel and find a list of hopefully not to many possible values for z.

Experimental parameters

The parameter K. As said earlier, we need to choose the parameter K big enough such that the small vector $(\mu_0, \ldots, \mu_{r-1})$ satisfies $\sum_{i=0}^{r-1} \mu_i V'_i = 0 \bmod N$. We notice experimentally that the value of K does not influence if our small vector is satisfying the condition or not. At worst, a bigger K seems to reduce the efficiency of our algorithm. For this reason and for simplicity sake, we choose $K = 1$.

The parameters r and t. This attack against a LCG needs $r + t - 1$ consecutive outputs. For a small ℓ, this attack needs more outputs than the one based on a Coppersmith method. But it allows us to recover the multiplier even when $\ell = n/2$. We present in the following table the parameters and computing times to find a polynomial f that satisfies $f(z) \equiv 0 \bmod N$. The experimental values are averages on a hundred instances of the algorithm.

ℓ/n	0.2	0.3	0.4	0.5	0.6
(r,t)	$(4,3)$	$(5,4)$	$(7,3)$	$(9,4)$	$(11,5)$
$r+t-1$	6	8	9	12	15
$n=32$	$0.0007s$	$0.0008s$	$0.0008s$	$0.001s$	$0.002s$
$n=64$	$0.0007s$	$0.001s$	$0.001s$	$0.003s$	$0.002s$
$n=1024$	$0.002s$	$0.004s$	$0.004s$	$0.005s$	$0.009s$

The size of the entries. The parameters K, r and t are chosen and we have constructed a polynomial f satisfying $f(z) \equiv 0 \bmod N$. But finding the polynomial f is only the first part of the algorithm and now we need to compute the roots of f. As $N = 2^n$ is highly composed, the number of roots of f is only bounded by 2^n. Experimentally, it happens quite regularly that f has too many roots and it prevents us to do massive tests or averages as we are easily stuck in bad cases. The following table will resume the different advantages and disadvantages of each attack against the LCG. All the real computing times are averages on ten instances of the algorithm with $n = 1024$.

ℓ/n	0.2	0.3	0.4	0.5
m for attack 1	3	3	6	*none*
m for attack 2	2×3	2×4	2×8	*none*
m for attack 3	6	8	9	12
computing time for attack 1	$0.02s$	$0.02s$	$2s$	*none*
computing time for attack 2	$0.2s$	$0.2s$	$15s$	*none*
computing time for attack 3	$1.5s$	$1.9s$	$1.4s$	$1.8s$

4 Attacks Against the Fast Knapsack Generator

Let $\omega_0, \ldots, \omega_{n-1}$ be n n-bits integers and let $u_0, u_1, \ldots$ be a sequence of bits generated by a linear feedback shift register (LFSR) over $\mathbb{F}_2$ with an irreducible characteristic polynomial of order n. At step i, the *knapsack generator* computes

$$v_i = \sum_{i=0}^{n-1} u_{i+j}\omega_j \mod 2^n$$

and outputs the leading $n - \ell$ bits of v_i.

This generator is defined by $n^2 + 2n$ bits: n bits for the public feedback polynomial, n bits for the initial control bits and n^2 bits for the weights.

The *fast knapsack generator* is a knapsack generator with special weights. We replace the arbitrary weights by $\omega_i = z^{n-i}y$, for y, z two integers of n bits. This new generator is defined by $4n$ bits (as we only need $2n$ bits for the weights) and faster. Instead of computing $v_{i+1} = \sum_{i=0}^{n-1} u_{i+1+j}\omega_j \mod 2^n$, with n additions, we directly compute

$$v_{i+1} = u_i zy + zv_i - u_{n+i}z^{n+1}y \mod 2^n \tag{2}$$

with only three additions. The control bits (u_i) come from a LFSR. Even if the LFSR is not cryptographically secure, as its characteristic polynomial is irreducible, we can assume the (u_i) follow a uniform distribution, at least from a statistical view point [14]. Hence the case where $v_{i+1} = zv_i \bmod 2^n$ (i.e. $u_i = u_{n+i} = 0$) appears with probability $\frac{1}{4}$. Then again we will need some notations.

- $H_i = (v_i \ quo \ 2^\ell) \times 2^\ell + 2^{\ell-1}$, where *quo* denotes the quotient of the Euclidean division (H_i is constructed from the output, hence it is known)
- $\delta_i = v_i - H_i$ (δ_i represents the ℓ discarded bits, it is unknown)
- m is the number of outputs we have.

The trick in this attack is to notice our PRNG behaves like a LCG in one iteration with probability 1/4. As we have two different algorithms to attack our specific LCG, we will have two different algorithms to attack the fast knapsack generator. These two attacks follow the same scheme: choosing when we are going to assume the PRNG behaves like a LCG, using an attack against the assumed LCG, obtain a z and some complete internal states, using the following outputs to guess the y and finally check the consistency.

4.1 Attack via Coppersmith method with consecutive outputs

Finding z: We choose $k+1$ consecutive outputs out of m, hence we choose k steps where we assume the PRNG acts as a LCG. On these $k+1$ outputs H_i's we apply the first algorithm we have against our specific LCG and obtain the δ_i's completing the $k+1$ chosen outputs (as $v_i = H_i + \delta_i$). If our assumptions is false, the δ_i returned by our Coppersmith method might not be integers. If it is the case, we start again with other sets of $k+1$ consecutive outputs until the δ_i are integers. Then we can complete our outputs to obtain $k+1$ complete consecutive internal states. Due to the use of a highly composite modulus 2^n, computing the z is not completely straightforward. If we know v_i and v_{i+1} such that $v_{i+1} = zv_i \bmod 2^n$ we might have to deal with a v_i non-invertible mod 2^n. But usually the 2-adic valuation of v_i does not exceed 5 so it is never a problem to do an exhaustive search on the possible values for z.

Finding y: Based on our first assumption, we know z and $k+1$ complete internal states of our PRNG. We call v_i our last known complete internal state and concentrate on v_{i+1} and v_{i+2}. Based on the structure of the PRNG, there is only 16 possibilities for the relations between v_i,v_{i+1} and v_{i+2}. If these relations are part of the 8 following possibilities, we can recover y again with a Coppersmith method using a lattice of dimension 4.

$$\begin{cases} v_{i+1} = zv_i + zy \mod 2^n \\ v_{i+2} = zv_{i+1} + zy \mod 2^n \end{cases} \quad \begin{cases} v_{i+1} = zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} = zv_{i+1} - z^{n+1}y \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} = zv_i + zy \mod 2^n \\ v_{i+2} = zv_{i+1} - z^{n+1}y \mod 2^n \end{cases} \quad \begin{cases} v_{i+1} = zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} = zv_{i+1} + zy \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} = zv_i + zy \mod 2^n \\ v_{i+2} = zv_{i+1} + zy - z^{n+1}y \mod 2^n \end{cases} \quad \begin{cases} v_{i+1} = zv_i + zy - z^{n+1}y \mod 2^n \\ v_{i+2} = zv_{i+1} + zy \mod 2^n \end{cases}$$

$$\begin{cases} v_{i+1} = zv_i - z^{n+1}y \mod 2^n \\ v_{i+2} = zv_{i+1} + zy - z^{n+1}y \mod 2^n \end{cases} \quad \begin{cases} v_{i+1} = zv_i + zy - z^{n+1}y \mod 2^n \\ v_{i+2} = zv_{i+1} - z^{n+1}y \mod 2^n \end{cases}$$

For example, we assume we are in the first case, hence

$$\begin{cases} v_{i+1} = zv_i + zy \mod 2^n \\ v_{i+2} = zv_{i+1} + zy \mod 2^n. \end{cases}$$

Subtracting the first equation to the second and replacing v_{i+1} by $H_{i+1}+\delta_{i+1}$ and v_{i+2} by $H_{i+2} + \delta_{i+2}$, we obtain:

$$H_{i+2} + \delta_{i+2} - H_{i+1} - \delta_{i+1} = zH_{i+1} + z\delta_{i+1} - zv_i \mod 2^n$$

(we recall that, at this point, v_i and z are assumed to be known). Hence $(\delta_{i+1}, \delta_{i+2})$ is a root of a polynomial in two variables of degree 1 $\mod 2^n$. It can be recovered thanks to a Coppersmith method. Once we have v_{i+1}, computing y is straightforward (once again, if the δ_i are not integers it means either our first assumption is false either the couple (v_{i+1}, v_{i+2}) is not of this form).

Remark 2 *There are several little optimisations/improvements we can do in this step. But it is mostly finding more particular cases so, for the sake of simplicity, we decided to not describe them here.*

Checking consistency: We have made a first assumption: the $k+1$ chosen outputs of our PRNG can be seen as truncated outputs of a LCG. We have made a second assumption (v_{i+1}, v_{i+2}) is of a chosen form between the eight listed possibilities. If y and z are the correct ones, we should be able to check consistency from one to the next (for example H_{i+3} should be given by one of the four following internal states: $zv_{i+2}, zy + zv_{i+2}, zv_{i+2} - z^{n+1}y$ or $zy + zv_{i+2} - z^{n+1}y$). If the consistency is not obtained, it means one of our assumption is false and we must either change our assumption on (v_{i+1}, v_{i+2}) if we did not explore the eight possibilities either start again from the beginning with a new set of consecutive outputs.

Analysis of the attack For a given k, we want to know m the number of outputs needed such that the probability of the PRNG acting as a LCG at least

k times in a row is greater than $1/2$. We have done the computation and we do not obtain a nice formula for m. The details are given in Appendix A and here we will only give the numerical result for $k = \{2, 3, 4, 5\}$.

Once m is greater than the computed bound, we hope there will be a set of $k+1$ consecutive outputs acting like a LCG. The two outputs following the last chosen one need to be in eight possibilities out of sixteen. Again it happens with probability $1/2$.

Hence, for a given k, the attack should work with probability greater than $1/4$ if m is greater than what is given in the table and $l/n < \binom{k}{2}/\Gamma(k)$ (as seen in Section 3). In this case we will have to run in the worst case $m-k$ instances of LLL on a lattice of dimension $k+1+3\binom{k}{2}$ and $8(m-k)$ instances of LLL on a lattice of dimension 4, each with entries of size n.

k	2	3	4	5
m	15	58	236	944
number of lattices $\leqslant$	13	55	232	939
dimension of lattices	6	13	23	36
ℓ/n (theoritical) $<$	1/7	3/16	6/29	5/23
ℓ/n (experimental) $\leqslant$	0.3	0.35	0.38	0.40
computing time for $n = 32$	$0.02s$	$0.11s$	$0.9s$	$10s$
computing time for $n = 64$	$0.02s$	$0.15s$	$2s$	$28s$
computing time for $n = 1024$	$0.04s$	$1.1s$	$31s$	$(950s)$

The computing time is an average of ten instances of the algorithm. When the algorithm becomes too slow to compute the average, an we give an estimation. The estimation comes from the execution time for one lattice multiplied by half the number of lattices. These specials cases are between parenthesis.

4.2 Attack via Coppersmith method with sparse outputs

Finding z We choose k outputs H_i out of $m-1$ outputs (we cannot choose the last one) and consider k pairs of outputs (H_i, H_{i+1}). It does not mean we work with $2k$ outputs as some pairs can overlap. On these k pairs of outputs we apply the second algorithm we have against our specific LCG and obtain δ_i's. If our assumption is false, the δ_i might not be integers. If it is the case, we start again with other sets of k pairs of outputs until the δ_i are integers. Then we can complete our outputs (as $v_i = H_i + \delta_i$) to obtain at most $2k$ complete consecutive internal states. Computing the z is not completely straightforward. If we know v_i and v_{i+1} such that $v_{i+1} = zv_i \mod 2^n$ we might have to deal with a v_i non-invertible mod 2^n. But usually the valuation 2-adic of v_i does not exceed 5 so it is never a problem to do an exhaustive search on the possible values for z.

The steps of *Finding y* and *Checking consistency* are the same as for the previous attack.

Analysis of the attack We want our PRNG to act at least k times like a LCG with probability greater than $1/2$. We suppose we clock our PRNG $m-1$ times (so we obtain m outputs). The probability that the PRNG acts as a LCG on one iteration is $1/4$. Hence we want k to be the strong *median* of a Binomial distribution of parameters $(m-1,\, 1/4)$. We consider the following theorem from [10].

Theorem 1. *If X is a $B(n,p)$, the median can be found by rounding off np to k if the following condition holds:*

$$|k - np| \leqslant \min(p, 1-p)$$

k is the strong median except when $p = 1/2$ and n is odd.

In our case where $p = 1/4$, we see that, given a k, the smaller number of trials satisfying this inequality is $4k-1$. Hence we choose $m = 4k$.

Once m is greater than $4k$, we hope our PRNG will act at least k times like a LCG. The two outputs following the last chosen one need to be in eight possibilities out of sixteen. Again it happens with probability $1/2$.

Hence, for a given k, the attack should work with probability greater than $1/4$ if m is greater than $4k$ and $l/n < (k-1)/4k$ (as seen in Section 3). In this case we will have to run in the worst case $\binom{4k}{k}$ instances of LLL on a lattice of dimension at worst $2k+3\binom{k}{2}$ and $8\binom{4k}{k}$ instances of LLL on a lattice of dimension 4, each with entries of size n.

k	2	3	4	5	6	7	8
m	8	12	16	20	24	28	32
number of lattices $\leq$	21	165	1365	11628	100947	888030	7888725
dimension of lattices $\leq$	7	15	26	40	57	77	100
ℓ/n (theoritical) $<$	1/8	1/6	3/16	1/5	5/24	3/14	7/32
ℓ/n (experimental) $\leqslant$	0.16	0.25	0.31	0.34	0.36	0.38	0.40
computing time for $n = 32$	$0.04s$	$0.6s$	$11s$	$(115s)$	$(2000s)$	$(11h)$	$(219h)$
computing time for $n = 64$	$0.03s$	$0.7s$	$21s$	$(230s)$	$(4000s)$	$(25h)$	$(548h)$
computing time for $n = 1024$	$0.06s$	$4s$	$(130s)$	$(4500s)$	$(22h)$	$(617h)$	$(1.6y)$

Again, the computing time is an average of ten instances of the algorithm running on the same laptop. When the algorithm becomes too slow to compute the average, we give an estimation. The estimation comes from the execution time for one lattice multiplied by half the number of lattices. These specials cases are between parenthesis.

Remark 3 *To computes these probabilities, we assumed we always had two outputs (v_{i+1}, v_{i+2}) following our output v_i. This is not always the case but this problem can be easily solved by choosing either another known v_i or the two preceding values of v_i instead of the following ones.*

Remark 4 *As the number of instances of LLL needed is $\binom{4k}{k}$, the computing time of the algorithm quickly explodes.*

4.3 Attack via Stern's attack on the LCG

Finding z: We choose $k+1$ consecutive outputs out of m, hence we choose k steps where we assume the PRNG acts as a LCG. On these $k+1$ outputs H_i's we apply the modified algorithm we have against our specific LCG and obtain a list of possible values for z. For each of these values, we are going to compute what we assume the internal states are. If we have the right value of z, then the vector of internal states $(v_i, \ldots, v_{i+k})$ is in the lattice spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & z & \ldots & z^k \\ 0 & 2^n & \ldots & 0 \\ & & \ddots & \\ 0 & 0 & 0 & 2^n \end{pmatrix}.$$

Also, this vector is close to the target vector $(H_i, \ldots, H_{i+k})$. Hence we use a CVP solver on our matrix and the target vector to find our vector of internal states.

Remark 5 *CVP stands for Closest Vector Problem. Given a lattice Λ and an arbitrary target vector T, a CVP solver outputs the closest vector to T which is in the lattice Λ.*

The steps of *Finding y* and *Checking consistency* are the same as for the previous attack.

4.4 Summary of our results

These computing times are averages on a hundred instances of the algorithm. As usual between parenthesis are the estimations given by the time for one lattice multiplied by half of the number of lattices.

– Attack via a Coppersmith method with consecutive outputs

ℓ/n	0.2	0.3	0.4	0.5
m	15	15	944	*none*
computing time for $n = 32$	$0.02s$	$0.02s$	$10s$	*none*
computing time for $n = 64$	$0.02s$	$0.02s$	$28s$	*none*
computing time for $n = 1024$	0.04	$0.04s$	$(950s)$	*none*

– Attack via a Coppersmith method with sparse outputs

ℓ/n	0.2	0.3	0.4	0.5
m	12	16	32	*none*
computing time for $n = 32$	$0.6s$	$11s$	$(219h)$	*none*
computing time for $n = 64$	$0.7s$	$21s$	$(548h)$	*none*
computing time for $n = 1024$	$4s$	$(130s)$	$(1.6y)$	*none*

– Attack via Stern's algorithm

ℓ/n	0.2	0.3	0.4	0.5
m	944	15138	60565	3876354
computing time for $n = 32$	$3.6s$	$(53s)$	$(300s)$	$(16h)$
computing time for $n = 64$	$11s$	$(227s)$	$(1200s)$	$(38h)$
computing time for $n = 1024$	$(700s)$	$(4h)$	$(11h)$	$(970h)$

The next generator we are going to analyse is based on constant-recursive sequences. These mathematical objects are completely linear. In the one hand it means they are fairly easy to manipulate. In the other hand it makes them very vulnerable to attacks. To hides the linear properties of its internal states, the generator uses two different moduli (as the reduction by two different moduli does not commute: $(a + b \bmod m_1) \bmod m_2$ tends to be different from $(a + b \bmod m_2) \bmod m_1$).

5 Combined Multiple Recursive Generators (CMRG)

These PRNGs output a linear operation between two or more congruential constant-recursive sequences over different moduli, pairwise coprime, of the same length. The coefficients of the sequences and the moduli are known, only the initial conditions are secret. We are going to focus on CMRG outputting the difference between two constant-recursive sequences of order three, (x_i) and (y_i) over two different moduli m_1 and m_2 of the same length n.

At step i, the generator computes

$$\begin{aligned} x_i &= a_{11}x_{i-1} + a_{12}x_{i-2} + a_{13}x_{i-3} \mod m_1 \\ y_i &= a_{21}y_{i-1} + a_{22}x_{i-2} + a_{23}y_{i-3} \mod m_2 \\ z_i &= x_i - y_i \mod m_1 \end{aligned}$$

and outputs z_i.

The values a_{11}, a_{12}, a_{13}, , a_{21}, a_{22}, a_{23}, m_1 and m_2 are known. The values x_0, x_1, x_2, y_0,y_1 and y_2 form the seed of the generator.

As m_1 and m_2 are coprime, by the Chinese Reminder Theorem we know that the sequences (x_i) and (y_i) are projections of a lifted constant-recursive sequence modulo m_1m_2 that we will call (X_i). This new sequence will be defined by $X_{i+3} = AX_{i+2} + BX_{i+1} + CX_i \bmod m_1m_2$ where A, B, C are given by:

$$\begin{aligned} A &\equiv a_{11} \bmod m_1 \text{ and } A \equiv a_{21} \bmod m_2 \\ B &\equiv a_{12} \bmod m_1 \text{ and } B \equiv a_{22} \bmod m_2 \\ C &\equiv a_{13} \bmod m_1 \text{ and } C \equiv a_{23} \bmod m_2 \end{aligned}$$

and the initial conditions X_0, X_1, X_2 in $\{0, \ldots, m_1m_2 - 1\}$ satisfy:

$$\begin{aligned} X_0 &\equiv x_0 \bmod m_1 \text{ and } X_0 \equiv y_0 \bmod m_2 \\ X_1 &\equiv x_1 \bmod m_1 \text{ and } X_1 \equiv y_1 \bmod m_2 \\ X_2 &\equiv x_2 \bmod m_1 \text{ and } X_2 \equiv y_2 \bmod m_2. \end{aligned}$$

The sequences (x_i) and (y_i) are given by $x_i = X_i \bmod m_1$ and $y_i = X_i \bmod m_2$.

5.1 Attack on the MRG32

Notations: We denote by z'_i the integer value $x_i - y_i$ which can be different from $z_i = x_i - y_i \bmod m_1$. As x_i is already in $\{0, \ldots, m_1 - 1\}$ and y_i is already in $\{0, \ldots, m_2 - 1\}$, we have that $z'_i = z_i$ or $z'_i = z_i - m_1$. We also denote by u the inverse of m_1 modulo m_2 ($um_1 \equiv 1 \bmod m_2$).

Proposition 1 *For every $i \geq 0$, $(x_i, x_{i+1}, x_{i+2}, x_{i+3})$ is a root modulo $m_1 m_2$ of*

$$P_i(v_i, v_{i+1}, v_{i+2}, v_{i+3}) = k_{i+3}m_1 + v_{i+3} - A(k_{i+2}m_1 + v_{i+2}) - B(k_{i+1}m_1 + v_{i+1}) - C(k_i m_1 + v_i)$$

where k_i is the only integer in $\{0, \ldots, m_2 - 1\}$ such that $k_i \equiv -z'_i u \bmod m_2$.

Proof. As $X_i \equiv x_i \bmod m_1$, there exists an integer k_i such that $X_i = k_i m_1 + x_i$. For the same reason, there exists an integer $\hat{k}_i$ such that $X_i = \hat{k}_i m_2 + y_i$. Hence

$$z'_i = x_i - y_i = \hat{k}_i m_2 - k_i m_1.$$

Thus $k_i \equiv -z'_i u \bmod m_2$. As X_i is in $\{0, \ldots, m_1 m_2 - 1\}$, then k_i is in $\{0, \ldots, m_2 - 1\}$. To obtain the polynomial P_i we need to remember that $X_{i+3} = AX_{i+2} + BX_{i+1} + CX_i \bmod m_1 m_2$.

We have established that (x_0, x_1, x_2, x_3) is a root modulo $m_1 m_2$ of

$$P_1(v_0, v_1, v_2, v_3) = k_3 m_1 + v_3 - A(k_2 m_1 + v_2) - B(k_1 m_1 + v_1) - C(k_0 m_1 + v_0)$$

and each of its coordinates is bounded by m_1.

If this root is the only small one, we can expect to retrieve it thanks to a Coppersmith method. But it tends not to be the case. We will consider Λ the lattice containing all the differences between two roots of P_1 modulo $m_1 m_2$. If the smallest vector v of Λ has its coordinates smaller than m_1, then the vector $(x_0, x_1, x_2, x_3) - v$ could be a smaller root of $P_1 \bmod m_1 m_2$ and our attack might not work.

The *Gaussian heuristic* "predicts" that if Λ is a full-rank lattice and C is a measurable subset of $\mathbb{R}^d$, then the number of points of $\Lambda \cap C$ is roughly $\mathrm{vol}(C)/\mathrm{vol}(\Lambda)$. In particular, this asserts that the norm of the shortest (non-zero) vector of Λ should be close to $\sqrt{d}\,\mathrm{vol}(\Lambda)^{1/d}$.

If we have two roots (x_0, x_1, x_2) and (x'_0, x'_1, x'_2) then

$$(x_3 - x'_3) - A(x_2 - x'_2) - B(x_1 - x'_1) - C(x_0 - x'_0) \equiv 0 \bmod m_1 m_2.$$

Hence the lattice Λ is spanned by the rows of the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & C \\ 0 & 1 & 0 & B \\ 0 & 0 & 1 & A \\ 0 & 0 & 0 & m_1 m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{4}(m_1m_2)^{1/4} \approx \sqrt{4}\times 2^{n/2} < \sqrt{4}\times 2^n \approx \sqrt{4}m_1$. Hence it is unlikely that (x_0, x_1, x_2, x_3) is the only root of P_1 modulo m_1m_2 such that each of its coordinates is bounded by m_1. We try to add other polynomials, hoping it will reduce the number of common roots.

If we consider the three polynomials P_1, P_2 and P_3, the lattice containing the difference between two commons roots will be spanned by the rows of the following matrix:

$$\begin{pmatrix} 1\,0\,0 & C & AC & BC + A^2C \\ 0\,1\,0 & B & C & B^2 + AC \\ 0\,0\,1 & A & (B + A^2) & C + 2AB + A^3 \\ 0\,0\,0 & m_1m_2 & 0 & 0 \\ 0\,0\,0 & 0 & m_1m_2 & 0 \\ 0\,0\,0 & 0 & 0 & m_1m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{6}(m_1m_2^3)^{1/6} \approx \sqrt{6}\times 2^n \approx \sqrt{6}m_1$. We are at the limit, we have no clear indication that the smallest vector of Λ is big enough. We cannot say that $(x_0, x_1, x_2, x_3, x_4, x_5)$ is the only common root of P_1, P_2 and P_3 modulo m_1m_2 such that each of its coordinates is bounded by m_1. Adding two polynomials was not enough. But the smallest difference between two common roots is far greater than before. So we keep adding polynomials.

If we consider the four polynomials P_1, P_2, P_3 and P_4, the lattice containing the difference between two commons roots will be spanned by the rows of the following matrix:

$$\begin{pmatrix} 1\,0\,0 & C & AC & BC + A^2C & C^2 + 2ABC + A^3C \\ 0\,1\,0 & B & C & B^2 + AC & 2BC + AB^2 + A^2C \\ 0\,0\,1 & A & (B + A^2) & C + 2AB + A^3 & 2AC + B^2 + 2A^2B + A^4 \\ 0\,0\,0 & m_1m_2 & 0 & 0 & 0 \\ 0\,0\,0 & 0 & m_1m_2 & 0 & 0 \\ 0\,0\,0 & 0 & 0 & m_1m_2 & 0 \\ 0\,0\,0 & 0 & 0 & 0 & m_1m_2 \end{pmatrix}.$$

Following the Gaussian heuristic, we can expect the shortest vector of this lattice to be of norm $\sqrt{7}(m_1m_2^4)^{1/7} \approx \sqrt{7} \times 2^{8n/7} > \sqrt{7} \times 2^n \approx \sqrt{7}m_1$. Hence $(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$ **is** likely to be the only common root of P_1, P_2, P_3 and P_4 modulo m_1m_2 such that each of its coordinates is bounded by m_1.

We can now describe the attack. From a_{11}, a_{12}, a_{13}, , a_{21}, a_{22} and a_{23} we construct A, B and C. Then we consider 7 outputs $z_0, \ldots, z_6$, and from them we guess $z'_0, \ldots, z'_6$ (we recall that $z'_i = z_i$ or $z'_i = z_i - m_1$). Now we have all the values we need to construct P_1, P_2, P_3 and P_4 as described in Proposition 1.

We use a Coppersmith method to find the only common root of P_1, P_2, P_3 and P_4 mod m_1m_2 with all of its coordinates bound by m_1. If we have correctly guessed the z'_i's, this root has to be $(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$, hence the initial

conditions we were searching for. Finally we check the consistency thanks to an eighth output.

Knowing the z_i's we have 2^7 set of possible values for the $z_i's$. For each set we run one instance of LLL on a lattice of dimension 12 (8 monomials + 4 polynomials) and entries of size n. So the time complexity is $\mathcal{O}(n^3)$.

5.2 The MRG32k3a by L'Écuyer

For this particular PRNG, the public values are $m_1 = 2^{32} - 209$, $m_2 = 2^{32} - 22853$, $a_{11} = 0$ $a_{12} = 1403580$, $a_{13} = 810728$, $a_{21} = 527612$, $a_{22} = 0$ and $a_{23} = 1370589$.

If we consider the four polynomials P_1, P_2, P_3, P_4 we find that the smallest difference between two common roots modulo $m_1 m_2$ is (-12600073455, 8717013482, 35458453228, 57149468535, 25239696855, -3505005772, 66309741613). We can see that each of its coordinates is greater than $2 \times m_1$, this ensures that (x_0, x_1, x_2, x_3, x_4, x_5, x_6) will be the only small common root of P_1,P_2,P_3 and P_4 modulo $m_1 m_2$. Our algorithm retrieves the initial conditions in 0.01 second with 8 outputs.

References

1. Benhamouda, F., Chevalier, C., Thillard, A., Vergnaud, D.: Easing Coppersmith methods using analytic combinatorics: Applications to public-key cryptography with weak pseudorandomness. In: Cheng, C.M., Chung, K.M., Persiano, G., Yang, B.Y. (eds.) PKC 2016, Part II. LNCS, vol. 9615, pp. 36–66. Springer, Heidelberg (Mar 2016). https://doi.org/10.1007/978-3-662-49387-8_3
2. Bouillaguet, C., Martinez, F., Sauvage, J.: Practical seed-recovery for the pcg pseudo-random number generator. IACR Transactions on Symmetric Cryptology **2020**(3), 175–196 (Sep 2020)
3. Ferrenberg, A.M., Landau, D.P., Wong, Y.J.: Monte carlo simulations: Hidden errors from "good" random number generators. Phys. Rev. Lett. **69**, 3382–3384 (Dec 1992)
4. Franke, D.: How i hacked hacker news (with arc security advisory). https://news.ycombinator.com/item?id=639976 (2009)
5. Frieze, A.M., Hastad, J., Kannan, R., Lagarias, J.C., Shamir, A.: Reconstructing truncated integer variables satisfying linear congruences. SIAM J. Comput. **17**(2), 262–280 (Apr 1988). https://doi.org/10.1137/0217016, http://dx.doi.org/10.1137/0217016
6. Frieze, A.M., Kannan, R., Lagarias, J.C.: Linear congruential generators do not produce random sequences. In: 25th FOCS. pp. 480–484. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715950
7. Von zur Gathen, J., Shparlinski, I.E.: Subset sum pseudorandom numbers: fast generation and distribution. Journal of Mathematical Cryptology **3**(2), 149–163 (2009)
8. Jochemsz, E., May, A.: A strategy for finding roots of multivariate polynomials with new applications in attacking RSA variants. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 267–282. Springer, Heidelberg (Dec 2006). https://doi.org/10.1007/11935230_18

9. Joux, A., Stern, J.: Lattice reduction: A toolbox for the cryptanalyst. Journal of Cryptology **11**(3), 161–185 (Jun 1998). https://doi.org/10.1007/s001459900042
10. Kaas, R., Buhrman, J.: Mean, median and mode in binomial distributions. Statistica Neerlandica **34**, 13–18 (1980)
11. Knellwolf, S., Meier, W.: Cryptanalysis of the knapsack generator. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 188–198. Springer, Heidelberg (Feb 2011). https://doi.org/10.1007/978-3-642-21702-9_11
12. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische annalen **261**(ARTICLE), 515–534 (1982)
13. L'Écuyer, P.: Random number generation with multiple streams for sequential and parallel computing. In: 2015 Winter Simulation Conference (WSC). pp. 31–44. IEEE (2015)
14. Mitra, A.: On the properties of pseudo noise sequences with a simple proposal of randomness test. International Journal of Electrical and Computer Engineering **3**(3), 164–169 (2008)
15. Ritzenhofen, M.: On efficiently calculationg small solutions of systmes of polynomial equations: lattice-based methods and applications to cryptography. Ph.D. thesis, Verlag nicht ermittelbar (2010)
16. Rueppel, R.A., Massey, J.L.: Knapsack as a nonlinear fonction. In: IEEE International Symposium on Information Theory. IEEE Press, NY (1985)
17. Stern, J.: Secret linear congruential generators are not cryptographically secure. In: 28th FOCS. pp. 421–426. IEEE Computer Society Press (Oct 1987). https://doi.org/10.1109/SFCS.1987.51
18. Van der Walt, S., Colbert, S.C., Varoquaux, G.: The NumPy array: A structure for efficient numerical computation. Computing in Science Engineering **13**(2), 22–30 (2011)
19. Wichmann, B.A., Hill, I.D.: Algorithm as 183: An efficient and portable pseudo-random number generator. Journal of the Royal Statistical Society. Series C (Applied Statistics) **31**(2), 188–190 (1982)

A Bernoulli trials

We suppose that we have n Bernoulli trials, each with a probability of success of p. We want to compute the probability of having a *run* of at least k consecutive successes. We denote this probability $Pr(n, p, k)$.

As we cannot have more successes than trials, if $k > n$ then $Pr(n, p, k) = 0$. If $k = n$, it means all the trials must be successes, hence $Pr(n, p, k) = p^k$.

If $n > k$ we have two excluding possibilities to have k successes. First possibility, a run of k successes happen in the last $n - 1$ trials. Second possibility, a run of k successes happen in the k first trial an there is *no* run of k successes in the last $n - 1$ trials. It means the first k trials are successes, then the $k + 1$-th trial is a fail and there is no run of k successes in the $n - k - 1$ remaining trials. Hence the probability of having a run of k successes in n trials when $n > k$ is $Pr(n, p, k) = Pr(n - 1, p, k) + p^k \times (1 - p) \times (1 - Pr(n - k - 1, p, k))$

We fix k and p and consider $S[n] = 1 - Pr(n, p, k)$. We notice that $(S[n])_{n\in\mathbb{N}}$ is a constant-recursive sequence:

$$S[n + 1] = S[n] - p^k(1 - p)S[n - k - 1]$$

of order $k + 1$ with initial terms being $S[0] = \dots S[k - 1] = 1$ and $S[k] = 1 - p^k$.

The explicit values of the sequence are given by $S[n] = C_1(r_1)^n + \dots + C_{k+1}(r_{k+1})^n$ where the r_i are the roots of the characteristic polynomial $x^{k+1} - x^k + p^k(1 - p)$ and the C_i are constants given by the initial terms.

In our case, we have m outputs and we want to know the probability of having $k + 1$ consecutive internal states of the form $v_{i+1} = zv_i \bmod 2^n$. Given a v_i, the probability that $v_{i+1} = zv_i \bmod 2^n$ is $1/4$. So our problem is to compute the probability of having a run of at least k successes in a sequence of $m - 1$ Bernoulli trials, the probability of success of each trial being $1/4$.

In the following table we give the minimal values of m such that the probability of having a run of k successes in $m - 1$ trials is greater than $1/2$.

k	2	3	4	5	6	7	8	11
m	15	58	236	944	3783	15138	60565	3876354

(Warning, these values are given by numerical approximations, they might not be exact.)

B Improvement of Coppersmith ?

Let P be the polynomial constructed thanks to the outputs of our LCG. We are searching for a root of P modulo N. In Section 2, we saw that we had two possibilities. We could directly construct the matrix used in the Coppersmith method $\mathcal{M}$ with only P or we could build a bigger set of polynomials P_i of the form $f = y_0^{k_0}, \dots, y_n^{k_n} P^{k_p}$. In Section 3, we presented attacks were the set of polynomials was not extended. The goal of this appendix will be to try to find

a family of polynomials P_i's such that we can retrieve the root even when more bits are discarded.

For the reader familiar with [1] by Benhamouda et al., we will use the same notations. We denote $\mathcal{P}$ the bigger set constructed from P. The polynomials in $\mathcal{P}$ are of the form $f = y_0^{k_0}, \ldots, y_n^{k_n} P^{k_p}$ and all linearly independent. We denote by $\chi_{\mathcal{P}}(f)$ the multiplicity of our small root as a root of f mod N: $\chi_{\mathcal{P}}(f) = k_p$. We denote $\mathfrak{M}$ the set of all the monomials appearing in $\mathcal{P}$. If m in $\mathfrak{M}$ is of the form $y_0^{k_0} \ldots y_n^{k_n}$, we denote $\chi_{\mathfrak{M}}(m) = k_0 + \cdots + k_n$. We know by equation (1) that the attack is suppose to work as long as

$$\ell/n \leq \frac{\sum_{f \in \mathcal{P}} \chi_{\mathcal{P}}(f)}{\sum_{m \in \mathfrak{M}} \chi_{\mathfrak{M}}(m)}$$

where ℓ is the number of discarded bits and n the size of the internal states of our generator.

B.1 Consecutive outputs

Here our Polynomial is $P = y_1^2 + 2H_1 y_1 + H_1^2 - y_0 y_2 - H_0 y_2 - H_2 y_0 - H_0 H_2$. We fix a parameter T and choose $\mathcal{P}_T$ as following:

$$\mathcal{P}_T = \{y_0^{k_0} y_1^{\epsilon} y_2^{k_2} P^{k_p} | \epsilon \in \{0,1\}, k_0 + \epsilon + k_2 + 2k_p \leq T\}$$

All the polynomials in $\mathcal{P}_T$ are linearly independent. Indeed, if we consider the monomial order $y_1 > y_0 > y_2$ then the leading monomial of $y_0^{k_0} y_1^{\epsilon} y_2^{k_2} P^{k_p}$ is $y_1^{2k_p+\epsilon} y_0^{k_0} y_2^{k_2}$ thus all leading monomials are different.

We are not going to precisely compute the set of monomial of $\mathcal{P}_T$ instead we are going to approach it with

$$\mathfrak{M}_T = \{y_0^{k_0} y_1^{k_1} y_2^{k_2} | k_0 + k_1 + k_2 \leq T\}.$$

Now we must compute $\sum_{f \in \mathcal{P}_T} \chi_{\mathcal{P}_T}(f)$ and $\sum_{m \in \mathfrak{M}_T} \chi_{\mathfrak{M}_T}(m)$:

$$\begin{aligned}
\sum_{f \in \mathcal{P}_T} \chi_{\mathcal{P}_T}(f) &= \sum_{k_0=0}^{T-2} \sum_{\epsilon=0}^{1} \sum_{k_2=0}^{T-2-k_0-\epsilon} \sum_{k_p=1}^{\lfloor \frac{T-k_0-\epsilon-k_2}{2} \rfloor} k_p \\
&= \lfloor \frac{((T+1)^2-1) \times ((T+1)^2-3)}{48} \rfloor \\
\sum_{m \in \mathfrak{M}_T} \chi_{\mathfrak{M}_T}(m) &= \sum_{k_0=0}^{T} \sum_{k_1=0}^{T-k_0} \sum_{k_2=0}^{T-k_0-k_1} k_0 + k_1 + k_2 \\
&= \frac{T(T+1)(T+2)(T+3)}{8}.
\end{aligned}$$

Thus this new construction should allow us to recover the small root as long as

$$\ell/n \leq \lfloor \frac{((T+1)^2-1) \times ((T+1)^2-3)}{48} \rfloor \times \frac{8}{T(T+1)(T+2)(T+3)}.$$

This value tends to 1/6.

To obtain a bound bigger than 1/7 (our already achieved result), we need $T \geq 13$. But $T = 13$ means our lattice would be of dimension 924, and running the LLL algorithm on a lattice of dimension 900 is hardly doable.

B.2 Sparse outputs

Here our Polynomial is $P = y_0 y_{i+1} - y_1 y_i + H_{i+1} y_0 + H_0 y_{i+1} - H_i y_1 - H_1 y_i + H_0 H_{i+1} - H_1 H_i$. We fix a parameter T and choose $\mathcal{P}_T$ as following:

$$\mathcal{P}_T = \{y_0^{k_0} y_1^{k_1} y_i^{k_i} P^{k_p} | k_0 + k_1 + k_i + 2k_p \leq T\} \bigcup \{y_1^{k_1} y_i^{k_i} y_{i+1}^{k_{i+1}} P^{k_p} | k_1 + k_i + k_{i+1} + 2k_p \leq T\}.$$

All the polynomials in $\mathcal{P}_T$ are linearly independent.

We are not going to precisely compute the set of monomial of $\mathcal{P}_T$ instead we are going to approach it with

$$\mathfrak{M}_T = \{y_0^{k_0} y_1^{k_1} y_i^{k_i} y_{i+1}^{k_{i+1}} | k_0 + k_1 + k_i + k_{i+1} \leq T\}.$$

Now we must compute $\sum_{f \in \mathcal{P}_T} \chi_{\mathcal{P}_T}(f)$ and $\sum_{m \in \mathfrak{M}_T} \chi_{\mathfrak{M}_T}(m)$:

$$\begin{aligned}
\sum_{f \in \mathcal{P}_T} \chi_{\mathcal{P}_T}(f) &= 2 \left(\sum_{k_0=0}^{T-2} \sum_{k_1=0}^{T-2-k_0} \sum_{k_i=0}^{T-2-k_0-k_1} \sum_{k_p=1}^{\lfloor \frac{T-k_0-k_1-k_2}{2} \rfloor} k_p \right) \\
&= \frac{(T+2)(2T^4 + 16T^3 + 28T^2 - 16T + 15 \times (-1)^T - 15)}{480} \\
\sum_{m \in \mathfrak{M}_T} \chi_{\mathfrak{M}_T}(m) &= \sum_{k_0=0}^{T} \sum_{k_1=0}^{T-k_0} \sum_{k_i=0}^{T-k_0-k_1} \sum_{k_{i+1}=0}^{T-k_0-k_1-k_i} k_0 + k_1 + k_i + k_{i+1} \\
&= \frac{T(T+1)(T+2)(T+3)(T+4)}{30}.
\end{aligned}$$

Thus this new construction should allow us to recover the small root as long as

$$\ell/n \leq \frac{(2T^4 + 16T^3 + 28T^2 - 16T + 15 \times (-1)^T - 15)}{T(T+1)(T+3)(T+4)} \times \frac{30}{480}.$$

This value tends to 1/8. But our second attack with one polynomial already recover the small root when $\ell/n \leq 1/8$. Hence adding more polynomials in our Coppersmith method does not seem relevant.