Practical seed-recovery for the PCG Pseudo-Random Number Generator

Charles Bouillaguet, Florette Martinez, Julia Sauvage

Predicting the PCG Pseudo-Random Number Generator In Practice

Charles Bouillaguet[1], Florette Martinez[2] and Julia Sauvage[3]

[1] Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
charles.bouillaguet@univ-lille.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
florette.martinez@lip6.fr
[3] Sorbonne Université
julia.sauvage@etu.upmc.fr

Abstract. The Permuted Congruential Generators (PCG) are popular conventional (non-cryptographic) pseudo-random generators designed in 2014. They are used by default in the `NumPy` scientific computing package. Even though they are not of cryptographic strength, their designer stated that predicting their output should be nevertheless be "challenging".
In this article, we present a practical algorithm that recovers all the hidden parameters and reconstructs the successive internal states of the generator. This enables us to predict the next "random" numbers, and output the seeds of the generator. We have successfully executed the reconstruction algorithm using 512 bytes of challenge input; in the worst case, the process takes 20 000 CPU hours.
This reconstruction algorithm makes use of cryptanalytic techniques, both symmetric and lattice-based. In particular, the most computationally expensive part is a guess-and-determine procedure that solves about 2^{52} instances of the Closest Vector Problem on a very small lattice.

Keywords: Pseudo-random number generator, guess-and-determine attack, truncated congruential generator, euclidean lattices, closest vector problem, practical attack

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> John von Neumann, 1949

1 Introduction

Pseudo-random generators (PRG) are well-studied primitives in symmetric cryptography. A PRG is an efficient deterministic algorithm that stretch a small random seed into a longer pseudo-random stream. To achieve cryptographic-grade pseudo-randomness, a PRG must ensure that the pseudo-random stream is computationally indistinguishable from a "truly" random sequence of bits by efficient adversaries. Alternatively, it is possible to define pseudo-randomness by asking that no efficient algorithm is capable of predicting the next pseudo-random bit with non-negligible accuracy. The two definitions are in fact equivalent.

```
u128 a;
u64 m = 0xda942042e4dd58b5;

u64_t lehmer64()
{
    a *= m;
    return a >> 64;
}
```

```
u32 a, b, c, d;


u32 xorshift128()
{
    u32 t = d;
    u32 s = a;
    d = c;
    c = b;
    b = s;
    t ^= t << 11;
    t ^= t >> 8;
    s ^= s >> 19;
    a = t ^ s;
    return a;
}
```

```
u64 a, b;


u64 xorshift128plus()
{
    u64 s1 = a
    u64 s0 = b;
    a = s0;
    s1 ^= s1 << 23;
    s1 ^= s1 >> 17;
    s1 ^= s0;
    s1 ^= s0 >> 26;
    b = s1;
    return a + b;
}
```

Figure 1: Some conventional pseudo-random generators, designed for speed and simplicity.

It is well-known that pseudo-random generators can be turned into symmetric encryption algorithm, by generating "random" masks to be used in the one-time pad. This is precisely what stream ciphers do.

Not all pseudo-random generators are of cryptographic strength. In some applications, it is simply not necessary: to be used in Monte-Carlo numerical simulations or generate random choices in games, a relaxed, non-cryptographic notion of pseudo-randomness may be sufficient. This allows for faster algorithms. For instance, `python` standard library's `random` module uses the Mersenne Twister [MN98]. The `C` library that comes along `gcc` (the `glibc`) uses a (poor) truncated linear congruential generator by default to implement the `rand` function.

In the realm of non-cryptographic random generators, a PRG is deemed "good enough" it is passes *some* efficient statistical tests — whereas the cryptographic notion of pseudo-randomness asks that it passes *any* efficient test. There are *de facto* statistical test suites; an initial battery of randomness tests for RNGs was suggested by Knuth in the 1969 first edition of *The Art of Computer Programming*. In 1996, Knuth's tests were then supplanted by Marsaglia's `Diehard` tests. In 2007, L'Ecuyer proposed the `TestU01` [LS07] library, whose "BigCrush" test is considered state-of-the-art by the relevant community, to the best of our knowledge. In 2010, the NIST proposed its own statistical test suite (Special Publication 800-22), to which improvements were later suggested [ZML+16]. We understand that the `PractRand` test suite also has a good reputation.

In any case, designers of conventional pseudo-random generators try to obtain the simplest and fastest algorithm that passes the day's favourite test suite. A few selected ones are shown in Fig. 1. `lehmer64` is a truncated linear congruential generator, touted as "the fastest PRNG that passes `BigCrush`" [Lem19]. `xorshift128` is a clever implementation of a 128-bit LFSR with period $2^{128} - 1$ due to Marsaglia [Mar03], using only a few simple 32-bit operations. `xorshift128+` is a improved version due to Vigna [Vig17] that returns the sum of two consecutive outputs of a Xorshift LFSR; it passes the "BigCrush" test suite and is the default PRNG in many Javascript implementations, including that in Google's V8 engine (Chrome), Firefox and Safari.

Failures in cryptographic pseudo-random generators have catastrophic security implications. Let us mention for instance the well-known problem in Debian Linux from 2008, where a bug in the OpenSSL package led to insufficient entropy gathering and to practical attacks on the SSH and SSL protocols (the only remaining source of entropy comes from the PID of the process, *i.e.* 16 bits or less of effective entropy) [YRS+09].

However, problems in non-cryptographic random number generators can also have dire consequences (barring the obvious case where they are used in lieu of their cryptographic counterparts). When they are used in scientific computing for Monte-Carlo methods, their defects have the potential to alter the results of numerical simulations. Ferrenberg et al. [FLW92] ran a classical Ferromagnetism Ising model Monte-Carlo simulation, in a special case where the exact results could be computed analytically, and compared the results of the simulation with the "true" answer. They used several conventional pseudo-random generators: a 32-bit linear congruential generator, two LFSRs, various combinations thereof, etc. They observed that changing the source of random numbers significantly altered the outcome of the numerical simulation. Different generators produced different biases: in particular a given LFSRs yielded energy levels that were systematically too low and critical temperatures that were always too high, while another kind of generator yielded the opposite (in many, repeated, trials).

The scientific computing community also realized that the need for fast *parallel* random number generation could be satisfied by the use of block ciphers in counter mode [SMDS11]. The need for speed then leads to the use of weakened cryptographic primitives (round-reduced AES or custom and presumably weak block-ciphers)

In most cases, it is fairly easy to see that a given conventional PRG does not meet the cryptographic notion of pseudo-randomness, and there are few exceptions. Most are fairly easy to predict, meaning that after having observed a prefix of the output, it is easy to produce the next "pseudo-random" bits. This makes a good source of exercises for cryptology students.

In this paper, we study the PCG family of non-cryptographic pseudo-random generators proposed by O'Neil [O'N14b, O'N14a]. She did not claim that the algorithm has cryptographic strength, but that predicting its output ought to be "challenging". We therefore took up the challenge.

PCG stands for "Permuted Congruential Generator": it essentially consists in applying a non-linear filtering function on top of a linear congruential generator (in a way reminiscent to the venerable filtered LFSRs). The resulting combination is fast and passes current statistical test suites. The PCG family contains many members, but we focus on the strongest one, named either PCG64 or PCG-XSL-RR. It has a 128-bit internal state and produces 64 bits when clocked. It is the default pseudo-random number generator in the popular NumPy [vCV11] scientific computing package for Python.

The internal state of the PCG64 generator is made of a 128-bit "state" and a 128-bit "increment", whose intended use is to provide several pseudo-random streams with the same seed (just as the initialisation vectors do in stream ciphers). A default increment is provided in case the end-user just want one pseudo-random stream with a single 128-bit seed.

Contribution. We describe an algorithm that reconstructs the full internal state of the strongest member of the PCG family. This allows to predict the pseudo-random stream deterministically and clock the generator backwards. The original seeds can also easily be reconstructed. The state reconstruction algorithm is practical and we have executed it in practice. It follows that predicting the output of the PCG should be considered practically feasible.

While the PCG pseudo-random generator is not meant as a cryptographic primitive, obtaining an actual prediction algorithm requires the use of cryptanalytic techniques. Making it practical requires in addition a non-trivial implementation effort.

Our algorithm reconstruct the internal state using a "guess-and-determine" approach: some bits of the internal state are guessed ; assuming the guesses are correct, some other information is computed ; a consistency check discards bad guesses early on ; then candidate internal states are computed and fully tested. The problem actually come in two distinct

flavors.

When the increment is known (for instance when it is the default value), a simplified prediction algorithm recovers the internal state from 192 bits of pseudo-random stream. The process runs in 20 CPU minutes. It guesses 37 bits of the internal state, then solves an instance of the Closest Vector Problem (CVP) in a 3-dimensional euclidean lattice. This requires about 50 arithmetic operations in total and reveals the entire internal state if the guesses are correct.

When the increment is unknown, things are a bit more complicated. This is the default situation in NumPy, where both the state and the increment are initialised using an external source of entropy. In this case, our prediction algorithm requires 4096 bits of pseudo-random stream ; it guesses between 51 and 55 bits, then for each guess it solves an instance of CVP in dimension 4 (using about 75 arithmetic operations). This recovers 64 more bits of information about the difference between two successive states, and this is enough to filter the bad guesses. This information can then be used in a subsequent and comparably inexpensive phase to recover the entire internal state. On average, the whole process requires a bit less than 20 000 CPU hours to complete.

We implemented our algorithms, then asked the designer of the PCG family to send us "challenge" pseudo-random streams ; we ran our code and emailed back the (correct) seeds used to generate the challenge streams the next day.

Related Work. Deterministic pseudo-random generators can be traced back to the work of Von Neumann and Metropolis on the ENIAC computer [vN51]; they suggested around 1946 to use the "middle-square" method: if u_n is a k-digit number, form u_{n+1} by taking the square of the $\frac{k}{2}$ middle digits of u_n. This is a venerable precursor of the Blum-Blum-Shub "provably secure" PRNG. The main problem of this method is that it produces sequences that quickly enters short cycles.

Lehmer later proposed linear congruential generators in 1949, also for use on the ENIAC computer [Leh49]. He gave the sequence defined by $u_0 = 47594118, u_{n+1} = 23u_n \bmod 10^8 + 1$ and proved that it had period 5882353, a clear improvement compared to the middle-square approach. More details on early pseudo-random generators can be found in [Knu98].

Knuth discussed whether *truncated* linear congruential generators could be good stream ciphers; he therefore studied the problem of recovering the internal state of a truncated linear congruential generator $x_{i+1} = ax_i + c \bmod 2^k$ when a and c are unknown [Knu85]; he gave an algorithm exponential in the number of truncated bits.

Boyar studied further the problem [Boy89] and presented an algorithm which could predict a linear congruential generator when all the parameters (multiplier, increment, modulus and initial state) are unknown; she extended her idea to the case of truncated linear congruential generators, under the condition that the number of bit unrevealed is really small in comparison to the size of the modulus.

Frieze *et al* [FHK+88] improved the efficiency of reconstruction algorithms in simpler cases. They supposed that the multiplier a and the modulus 2^k were known and used lattice-based techniques to recover a truncated linear congruential generator with more truncated bits.

Later on, Joux and Stern extended this result to the case where the multiplier a and the modulus 2^k are unknown, also using lattice techniques [JS98].

2 The PCG Pseudo-Random Number Generator Family

This section introduces some notations and describes the PCG64 non-cryptographic pseudo-random number generator (a.k.a. PCG-XSL-RR in the designer's terminology).

If $x \in \{0,1\}^n$ is an n-bit string, then $x[i{:}j]$ denotes the bit string $x_i x_{i+1} \dots x_{j-2} x_{j-1}$, where $x = x_0 \dots x_{n-1}$ (this is the "slice notation" used in Python). The set $\mathbb{Z}_{2^k}$ of integers modulo 2^k is seen as the set of k-bit strings. If x is a floating point number, then $\lfloor x \rceil$ denotes the nearest integer (using the "rounding half to even" tie-breaking rule — this is the default in IEEE754 arithmetic). If U is a vector or a sequence, then U_i is the i-th element (we use capital letters for these). If U is such a sequence, we denote by $U \bmod M$ the sequence $(U_0 \bmod M, U_1 \bmod M, \dots)$. The XOR operation is denoted $\oplus$, left and right rotations are denote $\lll$ and $\ggg$ respectively. Modular addition is denoted $+$ (or $\boxplus$ to make it even more explicit).

PCG64 has an internal state of 128-bit, which operate as a linear congruential generator modulo 2^{128}. More precisely:

$$S_{i+1} = aS_i + c \bmod 2^{128},$$

Where the "multiplier" a is a fixed 126-bit constant. The first initial state S_0 is the seed of the generator. The increment c may be specified by the user of the PRNG to produce different output streams with the same seed (just as the IV acts in a stream cipher). If no value of c is specified, then a default increment is provided. Note that c must be odd. The default values are:

$$\begin{aligned} a &= 47026247687942121848144207491837523525 && \text{(fixed)} \\ c &= 117397592171526113268558934119004209487 && \text{(default value, user-definable)} \end{aligned}$$

Each time the PRNG is clocked, 64 output bits are extracted from the internal state using a non-linear function that makes use of data-dependent rotations, in a way reminiscent of the RC5 block cipher [Riv94]. The six most significant bits of the internal state encode a number $0 \leq r < 64$. The two 64-bit halves of the internal state are XORed together, and this 64-bit result is rotated right by r positions.

The successive 64-bit outputs of the generator are $X_0, X_1, \dots$ where:

$$X_i = (\underbrace{S_i[0{:}64] \oplus S_i[64{:}128]}_{Y_i}) \ggg \underbrace{S_i[122{:}128]}_{r_i}. \tag{1}$$

For the sake of convenience, we denote by Y_i the XOR of the two halves of the state (before the rotation) and by r_i the number of shifts of the "i-th rotation".

Fig. 2 summaries the process. The overall design strategy is similar to that of a filtered LFSR: the successive states of a weak internal generator with a strong algebraic structure are "filtered" by a non-linear function.

Updating the internal state requires a $128 \times 128 \to 128$ multiplication operation. In fact, this can be done with three $64 \times 64 \to 128$ multiplication and two 64-bit additions. High-end desktop CPUs all implement these operations in hardware, so the generator is quite fast on these platforms.

3 Tools

In the rest of this paper, we often perform arithmetic operations on integers where only some bits are known. This leads to generation of unknown carries. If a, b are integers modulo 2^{128} and $0 \leq i < j < 128$, then there is a *carry* $0 \leq \gamma \leq 1$ (resp. a *borrow* $0 \leq \beta \leq 1$) such that:

$$(a \boxplus b)[i{:}j] = a[i{:}j] \boxplus b[i{:}j] \boxplus \gamma, \tag{2}$$

$$(a \boxminus b)[i{:}j] = a[i{:}j] \boxminus b[i{:}j] \boxminus \beta. \tag{3}$$

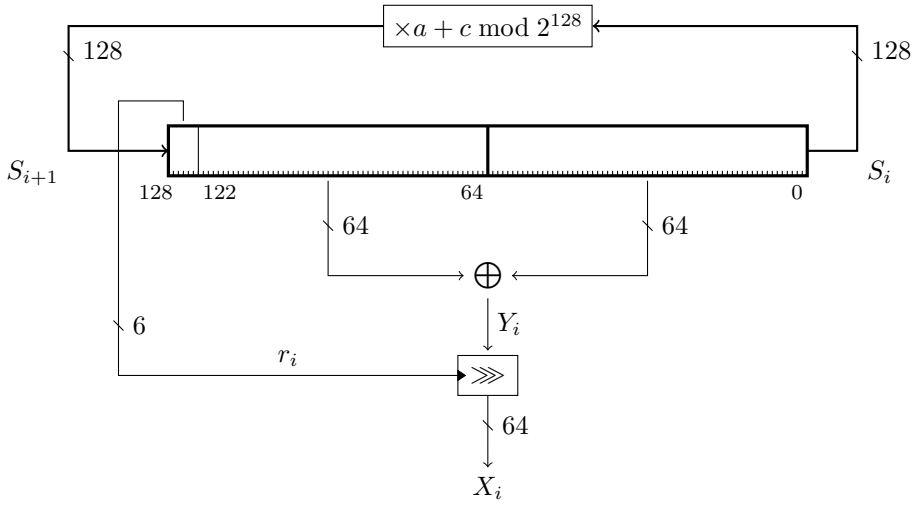

Figure 2: PCG64: Internal state update and output process.

3.1 Linear Congruential Generators and Lattices

Given an integer k, a fixed multiplier a, an increment c and a "seed" x, define the sequence:

$$U_0 = x, U_{i+1} = aU_i + c.$$

When reduced modulo 2^k, the sequence U form the successive states of a linear congruential generator (LCG). Let $\mathrm{LCG}_k(x, c)$ denote the vector $(u_0, u_1, u_2, \dots)$ of integers modulo 2^k. It is easy to check that:

$$\mathrm{LCG}_k(x + y, c + d) = \mathrm{LCG}_k(x, c) + \mathrm{LCG}_k(y, d), \tag{4}$$

$$\mathrm{LCG}_k(\lambda x, \lambda c) = \lambda\,\mathrm{LCG}_k(x, c). \tag{5}$$

Let $\mathcal{L}$ denote the euclidean lattice spawned by the rows of the following $n \times n$ matrix:

$$G_{n,k} = \begin{pmatrix} 1 & a & a^2 & \dots & a^{n-1} \\ 0 & 2^k & 0 & \dots & 0 \\ 0 & 0 & 2^k & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 2^k \end{pmatrix}$$

This lattice contains all n-terms geometric progressions of common ratio a modulo 2^k; therefore the first n terms of the sequence $\mathrm{LCG}_k(x, 0)$ give the coordinates of a vector in this lattice.

Reconstructing the state of a truncated linear congruential generator can generally be seen as the problem of finding a point of this lattice given only an approximation thereof, for instance when the least-significant bits of each components have been dropped. The "lattice approach" to truncated linear congruential generators is due to Frieze *et al.* [FHK+88].

Let U be a vector of n integers modulo 2^k such that $U_{i+1} = aU_i \bmod 2^k$; it is a geometric progression of common ratio a, modulo 2^k, therefore $(U_0, \dots, U_{n-1}) \in \mathcal{L}$. Let $T_i = U_i[k - \ell{:}k]$ denote the top ℓ bits of U_i, and let N denote an arbitrary "noise vector" such that $N_i \in \{-1, 0, 1\}$. Finally, set $\widetilde{T_i} = T_i + N_i \bmod 2^\ell$. We will be facing the following problem ("reconstructing noisy truncated geometric series") several times:

INPUT $\widetilde{T} = (\widetilde{T}_0, \ldots, \widetilde{T}_{n-1}) \in (\mathbb{Z}_{2^\ell})^n$, a (noisy) version of U truncated to the top ℓ bits.

OUTPUT $U_0 \bmod 2^k$, the first term of the (non-truncated) geometric sequence.

We will be facing a "high-dimension" instance in section 5.3 and many "low-dimension" in sections 4 and 5. The rest of this section discusses algorithmic tools to solve these problems. We first claim that $2^{k-\ell}\widetilde{T}$ is "close" to a point of the lattice $\mathcal{L}_{n,k}$.

Lemma 1. *There exists $U' \in \mathcal{L}$ such that $U'_i \equiv U_i \bmod 2^k$ and $\left\|U' - 2^{k-\ell}\widetilde{T}\right\| \leq 2\sqrt{n}2^{k-\ell}$.*

Proof. We first observe that U belongs to the lattice $\mathcal{L}$. We start by setting $U' \leftarrow U$, and we examine all coordinates of U:

- If $\widetilde{T}_i = T_i + N_i$ (without modulo), then we have:

$$\begin{aligned}\left|2^{k-\ell}\widetilde{T}_i - U_i\right| &= \left|2^{k-\ell}(T_i + N_i) - U_i\right| = \left|2^{k-\ell}N_i - U_i[0, k-\ell]\right| \\ &\leq \left|2^{k-\ell}N_i\right| + \left|U_i[0, k-\ell]\right| \leq 2^{k-\ell+1}.\end{aligned}$$

- Otherwise, there are two possible "wraparound" cases:
  - Either $T_i = 0$ and $N_i = -1$, which leads to $\widetilde{T}_i = 2^\ell - 1$. In this case, we have $U_i = U_i[0, k-\ell]$ and we set $U'_i = U_i + 2^k$ (note that this amount to adding a lattice vector to U', so U' stays in the lattice). We have:

$$\left|2^{k-\ell}\widetilde{T}_i - (U_i + 2^k)\right| = \left|2^{k-\ell} + U_i[0, k-\ell]\right| \leq 2^{k-\ell+1}.$$

  - Or $T_i = 2^\ell - 1$ and $N_i = +1$, which leads to $\widetilde{T}_i = 0$. This implies that $U_i = 2^k - 2^{k-\ell} + U_i[0, k-\ell]$; we set $U'_i = U_i - 2^k$ (again, with this modification U' stays in the lattice), and we find:

$$\left|2^{k-\ell}\widetilde{T}_i - (U_i - 2^k)\right| = \left|2^k - 2^{k-\ell} + U_i[0, k-\ell] - 2^k\right| \leq 2^{k-\ell+1}.$$

In the end, we have $\left\|U' - 2^{k-\ell}\widetilde{T}\right\| \leq 2\sqrt{n}2^{k-\ell}$, and $U' \equiv U \mod 2^k$. □

3.2 Reconstruction in "High" Dimension Using an Exact CVP Solver

Lemma 1 tells us that the approximation of a geometric sequence obtained by dropping least-significant bits cannot be arbitrarily far from a lattice point which reveals $U_0 \bmod 2^k$. Therefore, we may possibly reconstruct truncated geometric series by finding the lattice vector closest to the approximation we have. This means solving instances of the well-known Closest Vector Problem (CVP), a fundamental hard problem on lattices. It is NP-hard, and all known algorithms are exponential in the dimension of the lattice, yet they can be fairly practical up to dimension ≈ 70.

Let $\mathrm{CVP}(\mathcal{L}, x)$ denote the vector of $\mathcal{L}$ closest to the input vector x. Using the same notations as above, we want to know if $\mathrm{CVP}\left(\mathcal{L}, 2^{k-\ell}\widetilde{T}\right)$ is indeed U'. This will necessarily be the case when $\|2^{k-\ell}\widetilde{T} - U'\|$ is smaller than the length of the shortest non-zero vector of $\mathcal{L}$ — this quantity, the *first minimum* of the lattice, is denoted by $\lambda_1(\mathcal{L})$. By the triangular inequality, we have:

$$\left|\mathrm{CVP}(\mathcal{L}, 2^{k-\ell}\widetilde{T}) - U'\right| \leq \left|\mathrm{CVP}\left(\mathcal{L}, 2^{k-\ell}\widetilde{T}\right) - 2^{k-\ell}\widetilde{T}\right| + \left|2^{k-\ell}\widetilde{T} - U'\right|.$$

But, as the vector U' belongs to the lattice, by definition of the closest vector and by lemma 1:

$$|\operatorname{CVP}(\mathcal{L}, 2^{k-\ell}\widetilde{T}) - U'| \leq 2\left|2^{k-\ell}\widetilde{T} - U'\right| \leq 4\sqrt{n}2^{k-\ell}.$$

If we can prove that the right side of this inequality is smaller than the first minimum of the lattice $\lambda_1(\mathcal{L})$, then we would have proved that $\operatorname{CVP}(\mathcal{L}, 2^{k-\ell}\widetilde{T})$ indeed reveals $U_0 \bmod 2^k$.

In section 5.3, we will be facing the problem of reconstructing a geometric sequence modulo 2^{128} given arbitrarily many (noisy versions of the) most-significant 6 bits of successive elements of the sequence. Therefore we have $k = 128$ and $\ell = 6$, and we wish to determine the required number of samples, *i.e.* the value of n. This means finding the values of n such that $\sqrt{n}2^{124} \leq \lambda_1(\mathcal{L})$.

Starting from $n = \lceil 122/6 \rceil$, we computed the length of the shortest vector of the lattice spanned by $G_{n,128}$ for each successive n until the condition holds. The Shortest Vector Problem (SVP) is another well-known lattice NP-hard problem; we used the (almost) off-the-shelf `G6K` library [ADH+19], which gave results very quickly by sieving. `fplll` [dt16] was too slow above dimension 50, in the default settings.

After this computation, we found that the minimal possible n is 63: with $n = 63$, the shortest vector of $\mathcal{L}$ has length greater than $2^{127.02}$, which is high enough. This vector can be obtained by bootstrapping the geometric sequence with

$$U_0 = 121442528758503454790150022052419873 63$$

then reducing the terms modulo 2^{128} in zero-centered representation (subtracting 2^{128} to U_i if $U_i > 2^{127}$). It follows that when $n \geq 63, k = 128$ and $\ell = 6$, any CVP oracle will return a vector congruent to the original U when given $\widetilde{T}$.

3.3 Reconstruction in Low Dimension Using Babai's Rounding

In sections 4 and 5.1 we will need to reconstruct billions of noisy truncated geometric series modulo 2^{64} with very few terms, of which a large fraction of most-significant bits are known. In this setting, the CVP problem becomes much easier. This enables us to use faster and more *ad hoc* methods, such as Babai's rounding algorithm [Bab86].

If M is a square matrix, we denote by $|||M|||$ the induced matrix norm :

$$|||M||| = \sup_{x \in \mathbb{R}^n} \frac{\|xM\|}{\|x\|}$$

In the case of the $\|\cdot\|_2$ norm used throughout this paper, $|||M|||$ is the largest singular value of G; equivalently, it is the square root of the absolute value of the largest eigenvalue of $M^t M$.

Denote again by $\mathcal{L}$ the n-dimensional lattice spanned by the rows of $G_{n,64}$, and let H denote the LLL-reduction of $G_{n,64}$. The same lattice is also spanned by the rows of H. For instance, with $n = 3$:

$$H = \begin{pmatrix} -1241281756092 & 3827459685972 & -728312298332 \\ -5001120657083 & -2117155768935 & 5479732607037 \\ 8655886039732 & 3303731088004 & 6319848582548 \end{pmatrix}$$

Set $S = 2^{k-\ell}\widetilde{T}H^{-1}$; as $2^{k-\ell}\widetilde{T}$ is not *a priori* an element of the lattice, S need not be an integer vector. Let then R denote the rounding of S, *i.e.* $R_i = \lfloor S_i \rceil$. Then RH is an element of the lattice. Under the right conditions, it will be the vector of the lattice closest

to $2^{k-\ell}\widetilde{T}$. Indeed:

$$\begin{aligned}\|U' - RH\| &= \left\|U' - \left(R - 2^{k-\ell}\widetilde{T}H^{-1} + 2^{k-\ell}\widetilde{T}H^{-1}\right)H\right\| \\ &= \left\|U' - 2^{k-\ell}\widetilde{T} - \left(R - 2^{k-\ell}\widetilde{T}H^{-1}\right)H\right\| \\ &\leq \left\|U' - 2^{k-\ell}\widetilde{T}\right\| + \left\|R - 2^{k-\ell}\widetilde{T}H^{-1}\right\| \times |||H|||.\end{aligned}$$

By definition, R is the closest integer vector to $2^{k-\ell}\widetilde{T}H^{-1}$. But as U' is an element of the lattice, $U'H^{-1}$ is an integer vector. Thus $R - 2^{k-\ell}\widetilde{T}G^{-1}$ is shorter than $U'G^{-1} - 2^{k-\ell}\widetilde{T}G^{-1}$. Hence :

$$\begin{aligned}\|U' - RH\| &\leq \left\|U' - 2^{k-\ell}\widetilde{T}\right\| + \left\|R - 2^{k-\ell}\widetilde{T}H^{-1}\right\| \times |||H||| \\ &\leq \left\|U' - 2^{k-\ell}\widetilde{T}\right\| + \left\|U'H^{-1} - 2^{k-\ell}\widetilde{T}H^{-1}\right\| \times |||H||| \\ &\leq \left\|U' - 2^{k-\ell}\widetilde{T}\right\| + \left\|U' - 2^{k-\ell}\widetilde{T}\right\| \times |||H^{-1}||| \times |||H||| \\ &\leq \left\|U' - 2^{k-\ell}\widetilde{T}\right\| \times \left(1 + |||H^{-1}||| \times |||H|||\right).\end{aligned}$$

Note that $|||H^{-1}||| \times |||H|||$ is the condition number of the matrix H. Lattice reduction has the side effect of reducing the condition number, therefore it makes sense to use an LLL-reduced basis. If we can prove that the right side of the inequality is smaller than the first minimum of the lattice, then we would have proved that RH is indeed the closest vector we were searching for. Because we have fixed $k = 64$, by lemma 1 we have $\left\|2^{k-\ell}\widetilde{T} - U'\right\| \leq 2\sqrt{n}2^{64-\ell} - 1$. So, if we fix n we can search for the minimum number ℓ of known most-significant bits such that:

$$\left(1 + |||H||| \times |||H^{-1}|||\right) 2\sqrt{n}2^{64-\ell} \leq \lambda_1(\mathcal{L})$$

Table 1: minimal ℓ needed for a given n

n	$\|\|\|H\|\|\| \times \|\|\|H^{-1}\|\|\|$	$\lambda_1(\mathcal{L})$	minimum ℓ
3	2.87	$4.09e^{12} \simeq 2^{41.9}$	26
4	2.06	$2.44e^{14} \simeq 2^{47.8}$	20
5	3.77	$1.72e^{15} \simeq 2^{50.6}$	18
6	2.69	$1.03e^{16} \simeq 2^{53.2}$	15

When ℓ is greater than the values given in table 1, then Babai's rounding technique will always return the closest vector, and allow us to reconstruct a truncated geometric series.

3.4 Application to the `lehmer64` generator

Adapting the previous reasoning enables an efficient state reconstruction algorithm for the `lehmer64()` generator shown in Fig. 1. When clocked, it outputs the top 64 bits of a geometric sequence ($k = 128$ and $\ell = 64$). Three successive outputs are sufficient to reconstruct the internal state using Babai's rounding technique. This yields the following reconstruction algorithm:

```
def reconstruct(X):
    """
    Produce the internal state of the generator given three consecutive outputs of lehmer64().
    16 multiplications, 1 division, 11 additions and 3 roundings only.
    """
    a = 0xda942042e4dd58b5
```

```
r = round(2.64929081169728e-7 * X[0] + 3.51729342107376e-7 * X[1] + 3.89110109147656e-8 * X[2])
s = round(3.12752538137199e-7 * X[0] - 1.00664345453760e-7 * X[1] - 2.16685184476959e-7 * X[2])
t = round(3.54263598631140e-8 * X[0] - 2.05535734808162e-7 * X[1] + 2.73269247090513e-7 * X[2])
u = r * 1556524 + s * 2249380 + t * 1561981
v = r * 8429177212358078682 + s * 4111469003616164778 + t * 3562247178301810180
state = (a*u + v) % (2**128)
return state
```

4 State Reconstruction for PCG64 With Known Increment

We first consider the easier case where the "increment" (the c term in the definition of the underlying linear congruential generator) is known — recall that a default value is specified in case the user of the pseudo-random generator does not want to provide one.

In this case, reconstructing the 128-bit internal state S_i of the generator is sufficient to produce the pseudo-random flow with 100% accuracy (the generator can also be clocked backwards if necessary, so that the seed can be easily reconstructed). We therefore focus on reconstructing S_0 (the seed) from $X_0, X_1, X_2, \ldots$. A very simple strategy could be the following:

1. Guess the 64 upper bits of S_0 (this includes the rotation).
2. Compute the missing 64 lower bits using (1), with:

$$S_0[0{:}64] = S_0[64{:}128] \oplus (X_0 \lll S[122{:}128]).$$

3. Compute S_1 then extract X_1; if X_1 is correct, then output S_0.

This "baseline" procedure requires 2^{64} iterations of a loop that does a dozen arithmetic operations; it always output the correct value of S_0, and may output a few other ones (they can be easily discarded by checking X_2). An improved "guess-and-determine" state reconstruction algorithm is possible, which essentially amounts to expose a truncated version of the underlying linear congruential generator, and attack it using the tools exposed in section 3. This is possible by combining the following ingredients:

- The underlying linear congruential generator uses a power-of-two modulus, therefore the ℓ low-order bits of S_{i+1} are entirely determined by the ℓ low-order bits of S_i. More precisely, we have:

$$S_{i+1} = aS_i + c \bmod 2^\ell, \qquad \text{for all } 0 \leq \ell \leq 128 \tag{6}$$

  Therefore, guessing the least-significant bits of S_0 yields a "long-term advantage" that holds for all subsequent states.

- Guessing a 6-bit rotation r_i gives access to Y_i (the XOR of the two halves of the internal state). Thus, if a part of the state is known, then this transfers existing knowledge to the other half.

In figure 3, we see that guessing $S_0[0{:}\ell]$ and a few 6-bit rotations r_i give access to $S_i[58{:}64+\ell]$ for the corresponding states. Therefore, looking at $S_i[\ell{:}64+\ell]$, we are facing a truncated linear congruential generator on 64 bits, where we have access to the most $6+\ell$ bits of each state (denoted by T), for a few consecutive states. This is sufficient to reconstruct entirely the successive states of this truncated linear congruential generator. This reveals $S_0[\ell{:}64+\ell]$, and using (1) the entire S_0 can be reconstructed. The precise details follow.

We consider the sequence of internal states $S = (S_0, S_1, \ldots) = \mathrm{LCG}_{128}(S_0, c)$. We will guess the ℓ least-significant bits of S_0, therefore let us assume that their value is known and denote it by w. We define $S' = \mathrm{LCG}_{128}(S_0 - w, 0)$ and $K = \mathrm{LCG}_{128}(w, c)$ — this

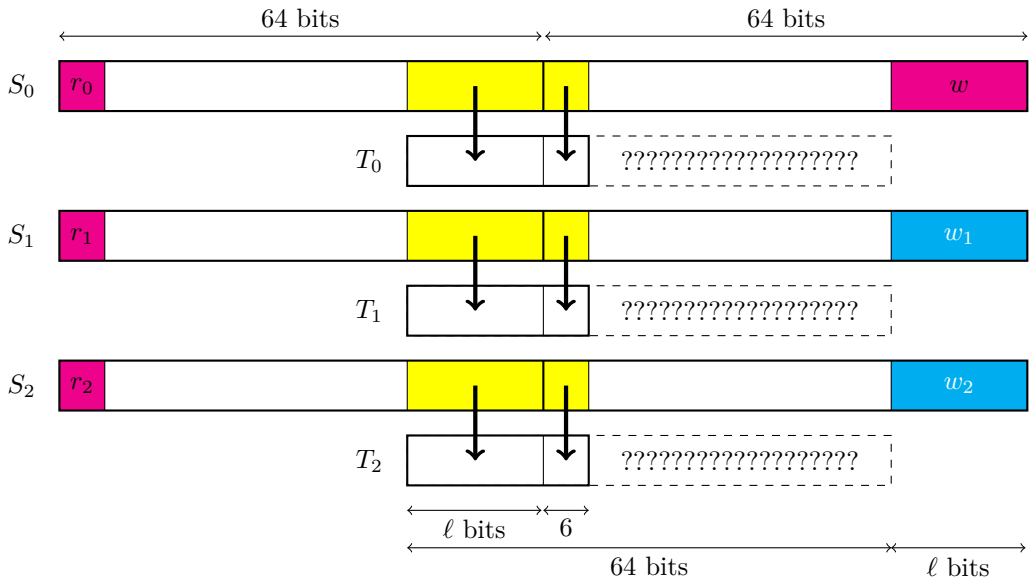

Figure 3: A guess-and-determine algorithm to reconstruct the first internal state S_0. Magenta bits are guessed; cyan bits are obtained using the linear congruence relation (6) modulo 2^ℓ; yellow bits are obtained from the output and the guessed rotations using (1).

is Known. By (4), we have $S' = S - K$. The point is that the elements of S' follow a geometric progression of common ratio a; in addition, the ℓ least significant bits of each components are equal to zero. It follows that $S'[\ell{:}64+\ell]$ also follows a geometric progression of common ratio a, this time modulo 2^{64}. The crux of the reconstruction algorithm is to find $S'[\ell{:}64+\ell]$.

We know $K_i[58{:}64+\ell]$ for all i, and for each guessed rotation r_i we have access to $T_i \overset{def}{=} S_i[58{:}64+\ell]$. We want $T'_i \overset{def}{=} (S_i \boxminus K_i)[58{:}64+\ell]$, which is the truncation of S'_i. Thanks to (3), we know that there is an unknown vector B of *borrows*, whose components are either 0 or 1, such that $T' = S[58{:}64+\ell] \boxminus K[58{:}64+\ell] \boxminus B$. Because the borrows are unknown, we in fact compute $\widetilde{T'} = S[58{:}64+\ell] \boxminus K[58{:}64+\ell]$, and clearly $\widetilde{T'} = T' \boxplus B$. We are thus in the context of the problem discussed in section 3, namely reconstructing a geometric sequence given $6+\ell$ (noisy) most-significant bits. The "noise" is the unknown vector B of borrows.

We will guess n rotations and ℓ least-significant bits of the state, for a total of $2^{6n+\ell}$ guessed bits. Table 1 gives a lower-bound on ℓ given n, and we see that the total number of guessed bits reaches a minimum of 38 when $n = 3$ and $\ell = 20$. Therefore, success is guaranteed if we guess $\ell = 20$ low-order bits of the state and three consecutive rotations.

The algorithm that reconstructs the internal state of the PCG64 generator with known increment proceeds as shown in algorithm 1. The point is that when the guesses are correct, then from the truncated geometric series T', the solution of the CVP instance reveals $U_j = S'_j[\ell{:}64+\ell]$. From there, the correction of the algorithm is easily established.

The procedure is completely practical. More details are given in section 6. Let us just mention that the procedure often works (twice faster) with $\ell = 19$ or even four times faster with $\ell = 18$ (with a reduced success probablity).

Algorithm 1 State reconstruction Algorithm (case where c is known)

procedure ReconstructState$_\ell(X_0, X_1, X_2)$
  // Statement involving j must be repeated for $j = 0, 1, 2$.
  $H \leftarrow$ LLL reduction of $G_{3,64}$
  $\ell \leftarrow 20$
  for $0 \le w < 2^\ell$ **do** ▷ Guess least-significant bits of S_0
    $K_j \leftarrow a^j w + c(a^j - 1)(a-1)^{-1} \bmod 2^{128}$ ▷ Known part
    for $0 \le r_0, r_1, r_2 < 64$ **do** ▷ Guess rotations
      $Y_j \leftarrow X_j \lll r_j$ ▷ Undo rotations
      $T_j \leftarrow (r_j \oplus Y_j[58{:}64]) + 64 \cdot (K_j \oplus Y_j)\,[0{:}\ell]$ ▷ Truncated LCG output
      $\widetilde{T}'_j \leftarrow T_j \boxminus K_j[58{:}64 + \ell]$ ▷ Truncated geometric series on $6 + \ell$ bits
      $(U_0, U_1, U_2) \leftarrow \left\lfloor 2^{58-\ell} \cdot (\widetilde{T}'_0, \widetilde{T}'_1, \widetilde{T}'_2) \cdot H^{-1} \right\rceil \cdot H$ ▷ CVP (Babai rounding)
      $S_0[0{:}64] \leftarrow K_0[0{:}64] + 2^\ell \cdot U_0[0{:}64 - \ell]$ ▷ Reconstruct S_0
      $S_0[64{:}128] \leftarrow S_0[0{:}64] \oplus Y_0$
      $S_1 \leftarrow aS_0 + c$ ▷ Recompute X_1
      $\widehat{Y}_1 = S_1[0{:}64] \oplus S_1[64{:}128]$
      if $\widehat{Y}_1 = Y_1$ **then** ▷ Check consistency
        output S_0 as a candidate internal state.

5 State Reconstruction for PCG64 With Secret Increment

The algorithm of section 4 does not apply directly to the general case where the value of c is unknown. A "baseline" procedure would consist in guessing $S_0[64{:}128]$ and $S_1[64{:}128]$; using eq. (1), this would reveal S_0 and S_1; from there, the increment c is easily to obtain, and every secret information has been reconstructed. This would take 2^{128} iterations of a very simple procedure, which is completely infeasible.

Set $\Delta S_i = S_{i+1} \boxminus S_i$; it is easily checked that ΔS_i is a geometric progression of common ratio a. Therefore, reconstructing both S_0 and ΔS_0 is sufficient to compute all subsequent states (and recover the unknown increment c). The global "guess-and-determine" strategy is essentially the same as before: gaining access to a truncated version of ΔS_i, solving a small SVP instance, reconstructing ΔS_0, then checking consistency.

Let us set:

$$\nabla S_i \stackrel{def}{=} S_i - S_0 \equiv \sum_{j=0}^{i-1} \Delta S_j \equiv \Delta S_0 \cdot \sum_{j=0}^{i-1} a^j \equiv \Delta S_0 \frac{a^i - 1}{a - 1} \mod 2^{128} \tag{7}$$

Note that $\nabla S_0 = 0$ and $\nabla S_1 = \Delta S_0$. Therefore, knowledge of Δ_0 entails that of the whole sequence of ∇S_i. The prediction algorithm we propose proceeds in three phases:

1. Reconstruct $\Delta S_0[0{:}64 + \ell]$ from $X_0, \ldots, X_4$, check consistency with $X_5, \ldots, X_{63}$.
2. Reconstruct all rotations r_i from this partial knowledge.
3. Fully reconstruct ΔS_0 from the rotations.
4. Reconstruct S_0 from ΔS_0 and the rotations.

Only the first phase is computationally intensive. The four steps are discussed in the next four subsections.

5.1 Partial Difference Reconstruction

In order to access to a part of ΔS_i, we use the same "guess-and-determine" strategy as in section 4: we guess the least significant bits of S_0 and some rotations, then check

consistency. The difference is that, since c is unknown, we must in addition guess the least significant bits of c to obtain the same "long-term advantage" (c is always odd; this makes one less bit to guess). We must also guess $k+1$ successive rotation to get information on k successive differences ΔS_i.

Confirming that the guesses are correct is less immediate. When c was known, we could reconstruct the internal state; from there, filtering out the bad guesses was easy. When c is unknown, the same strategy does not work, but a very strong consistency check can still be implemented.

We consider again the sequence of internal states $S = (S_0, S_1, \dots) = \mathrm{LCG}_{128}(S_0, c)$. We will guess the ℓ least-significant bits of S_0 and of c, therefore let us assume that their value is known and denote it by w_0 and c_0. We define $S' = \mathrm{LCG}_{128}(S_0 - w, c - c_0)$ and $K = \mathrm{LCG}_{128}(w_0, c_0)$ — again, K is Known and $S' = S - K$. This time, the components of S' do *not* follow a geometric progression; but we still have that the ℓ least significant bits of each S'_i are zero. Set $\Delta S'_i \stackrel{def}{=} S'_{i+1} - S'_i$; $\Delta S'[\ell{:}64+\ell]$ follows a geometric progression of common ratio a modulo 2^{64} (again). This time, we have to find $\Delta S'_0[\ell{:}64+\ell]$.

As in section 4, we have access to $T_i \stackrel{def}{=} S_i[58{:}64+\ell]$. We want to subtract the known part to obtain $T'_i \stackrel{def}{=} (S_i \boxminus K_i)[58{:}64+\ell]$, which is the truncation of S'_i. This again introduces an unknown vector B of *borrows*, and in fact we can only compute $\widetilde{T'} = S[58{:}64+\ell] \boxminus K[58{:}64+\ell]$, with $\widetilde{T'} = T' \boxplus B$. As explained above, to access a geometric sequence, we would like to obtain $\Delta T'_i \stackrel{def}{=} T'_{i+1} - T'_i$, but we can only compute:

$$\Delta\widetilde{T}'_i \stackrel{def}{=} \widetilde{T}'_{i+1} - \widetilde{T}'_i = (T'_{i+1} \boxminus T'_i) \boxplus (B_{i+1} \boxminus B_i)$$

We are thus still in the context of the problem discussed in section 3, but this time the "noise" caused by the carries is given by $B_{i+1} - B_i$. When the guesses are correct, then Babai's rounding will reconstruct $\Delta\widetilde{S}'[\ell{:}64+\ell]$ from $\Delta\widetilde{T}'$. This in turn yields $\Delta S_0[0{:}64+\ell]$.

Once we have found $\Delta S_0[0{:}64+\ell]$, we can compute $\nabla S_i[0{:}64+\ell]$ for any i because eq. (7) holds modulo $2^{64+\ell}$; because we have guessed the first rotation and the ℓ least significant bits of the state, using (1) we gain access to $S_0[58{:}64+\ell]$; combined with the "differences" ∇S_i, this reveals $S_i[58{:}64+\ell]$ for any i (and we already had $S_i[0{:}\ell]$). This allows us to compute $Y_i[0{:}\ell] = S_i[0{:}\ell] \oplus S_i[64{:}64+\ell]$ for any i. Given a "fresh" output X_i, and assuming that the guesses are correct, then we should have:

$$S_i[0{:}\ell] \oplus S_i[64{:}64+\ell] = (X_i \lll r_i)[0{:}\ell]. \tag{8}$$

In particular, if the guesses were correct, then we should have for any i:

$$S_i[0{:}\ell] \oplus S_i[64{:}64+\ell] \in \big\{(X_i \lll r)[0{:}\ell] \mid 0 \le r < 64\big\}. \tag{9}$$

If none of the 64 possible rotations yields a match, then the guesses made beforehand have to be wrong. As a consequence, bad guesses can be filtered with an arbitrarily low probability of false positives, by trying several indices i.

A few details still need to be fleshed out. To be precise, let us assume that we have guessed the ℓ least-significant bits of S_0 (we denote them by w_0) and the first rotation r_0. Set $Y_0 = X_0 \lll r_0$. We obtain the i-th state by $S_i \equiv \nabla S_i \boxplus S_0$; however, because the "middle" of S_0 is unknown, then an unknown *carry* may cross the 64-th bit during the addition and perturb $S_i[64{:}64+\ell]$. As a result, there is an unknown vector C, whose components are either 0 or 1, such that such that:

$$S_i[64{:}64+\ell] = C_i \boxplus \nabla S_i[64{:}64+\ell] \boxplus \underbrace{(w_0 \oplus Y_0[0{:}\ell])}_{S_0[64{:}64+\ell]}$$

Algorithm 2 Partial difference reconstruction algorithm (when c is unknown).

procedure ConsistencyCheck($\Delta S_0, w_0, Y_0, X_5, \ldots, X_k$)
  $v_0 = w_0 \oplus Y_0[0{:}\ell]$ ▷ $v_0 = S_0[64{:}64+\ell]$
  for $i = 5, \ldots, k$ **do**
    $u_i \leftarrow \Delta S_0(a^i - 1)(a-1)^{-1} \bmod 2^{64+\ell}$ ▷ $u_i = \nabla S_i[0{:}64+\ell]$
    $w_i = w_0 \boxplus u_i[0{:}\ell]$ ▷ $w_i = S_i[0{:}\ell]$
    $v_i = v_0 \boxplus u_i[64{:}64+\ell]$ ▷ $S_i[64{:}64+\ell] \in \{v_i, v_i'\}$
    $v_i' = v_i \boxplus 1$
    $\mathcal{C}_i \leftarrow \{w_i \oplus (X_i \lll r_i)[0{:}\ell] \mid 0 \le r_i < 64\}$ ▷ Check eq. (9)
    if $\{v_i, v_i'\} \cap \mathcal{C}_i = \emptyset$ **then**
      return False ▷ Bad Guesses
  return True ▷ No inconsistency

procedure ReconstructPartialDifference($X_0, \ldots, X_k$)
  // Statement involving j must be repeated for $j = 0, 1, 2, 3, 4$.
  $H \leftarrow$ LLL reduction of $G_{4,64}$
  $\ell \leftarrow 14$
  for $0 \le w_0 < 2^\ell$ and $0 \le c_0 < 2^{\ell-1}$ **do** ▷ Guess least-significant bits
    $K_j \leftarrow a^j w_0 + (2c_0+1)(a^j-1)(a-1)^{-1} \bmod 2^{128}$ ▷ Known part
    for $0 \le r_0, r_1, r_2, r_3, r_4 < 64$ **do** ▷ Guess rotations
      $Y_j \leftarrow X_j \lll r_j$ ▷ Undo rotations
      $T_j \leftarrow (r_j \oplus Y_j[58{:}64]) + 64 \cdot (K_j \oplus Y_j)\,[0{:}\ell]$ ▷ Truncated LCG
      $\widetilde{T}_j' \leftarrow T_j \boxminus K_i[58{:}64+\ell]$ ▷ Cancel known part
      $\Delta\widetilde{T}_j' = \widetilde{T}_{j+1}' \boxminus \widetilde{T}_j'$ ▷ Difference (truncated geom. seq.)
      $(\Delta U_0, \ldots, \Delta U_3) \leftarrow \left\lfloor (\Delta\widetilde{T}_0', \ldots, \Delta\widetilde{T}_3') \cdot 2^{58-\ell} \cdot \widetilde{H}^{-1} \right\rceil \cdot \widetilde{H}$ ▷ CVP
      $\Delta S_0[0{:}64+\ell] \leftarrow (K_1 \boxminus K_0)\,[0{:}\ell] + 2^\ell \cdot \Delta U_0[0{:}64]$ ▷ Check
      if ConsistencyCheck($\Delta_0, w_0, Y_0, X_5, \ldots, X_k$) **then**
        return $(w_0, c_0, r_0, \ldots, r_4, \Delta S_0)$.

In algorithm 2, ConsistencyCheck uses eq. (9) combined with this observation to discard bad guesses.

The heart of the algorithm is again the reconstruction of a truncated geometric progression. Looking at table 1, we see that the best choice consists in guessing 5 consecutive rotations and $\ell = 14$ least-significant bits. Therefore, ReconstructPartialDifference does 2^{57} iterations of the inner loop, and succeeds deterministically.

5.2 Predicting all the Rotations

Knowing the values of $\Delta S_0[0{:}64+\ell]$ as well as the ℓ least-significant bits of S_0 and c is sufficient to get rid the nastier feature PCG64: armed with this knowledge, we can determine all the subsequent rotations deterministically, at negligible cost, using eq (8). For each index i, it suffices to try the 64 possible values of r_i; only one should satisfy eq (8). The complete pseudo-code is shown in algorithm 3.

It is unlikely that several possible values of r_i match: each value is "checked" on ℓ bits, so an accidental match happens with probability $2^{\ell-6}$. The total number of lists returned by ReconstructRotations then follows a binomial distribution of parameters $2^{\ell-6}, k$. With $\ell = 14$ and $k = 64$, then only one rotation vector should pass the test for $0 \leq i < 64$ on average.

Algorithm 3 Rotations and full difference reconstruction algorithm

function ReconstructRotations($\Delta S_0, v_0, i, k$)
  // Return a list of potential $[r_i, r_{i+1}, \ldots, r_k]$; *assume that* $v_0 = S_0[64{:}64+\ell]$
  if $i > k$ **then**
    return [] ▷ End recursion
  $\mathcal{T} \leftarrow$ ReconstructRotations($\Delta S_0, v_0, i+1, k$) ▷ Find all the $(r_{i+1}, \ldots, r_k)$
  $\mathcal{H} \leftarrow$ [] ▷ List of possible r_i's
  $u_i \leftarrow \Delta S_0 (a^i - 1)(a-1)^{-1} \bmod 2^{64+\ell}$ ▷ $u_i = \nabla S_i[0{:}64+\ell]$
  $w_i = w_0 + u_i[0{:}\ell] \bmod 2^\ell$ ▷ $w_i = S_i[0{:}\ell]$
  $v_i = v_0 + u_i[64{:}64+\ell] \bmod 2^\ell$ ▷ $S_i[64{:}64+\ell] \in \{v_i, v_i'\}$
  $v_i' = v_i + 1 \bmod 2^\ell$
  for $0 \leq r < 64$ **do** ▷ Try all rotations
    if $w_i \oplus (X_i \lll r)[0{:}\ell] \in \{v_i, v_i'\}$ **then** ▷ Check eq. (8)
      $\mathcal{H} \leftarrow r :: \mathcal{H}$ ▷ New candidate r_i
  return $\{h :: t \mid h \in \mathcal{H}, t \in \mathcal{T}\}$ ▷ Return $\mathcal{H} \times \mathcal{T}$

5.3 Full Difference Reconstruction

Using $X_0, X_1, \ldots, X_{63}$, we recover all rotations and thus we recover the 6 most-significant bits of $S_0, S_1, \ldots, S_{63}$. This allows us to compute the 6 most significant bits of the differences ΔS_i between consecutive states (up to missing carries), and we are faced with the problem of reconstructing a 128-bit geometric progression using 63 consecutive outputs truncated to their 6 most-significant bits. There is again an unknown vector of borrows B such that $\Delta S_i[122{:}128] \boxplus C_i = r_{i+1} \boxminus r_i$.

Reconstructing ΔS_0 from the r_i is exactly the problem discussed in section 3.2. This can be done by solving an instance of CVP in dimension 63. We use the off-the-shelf CVP solver embedded in `fplll`: it runs in no measurable time.

5.4 Complete State Reconstruction

Once all the rotations have be recovered and ΔS_0 has been found entirely, the only thing that remain is to actually find the entire S_0. For this, we use again eq. (1), coupled with

the "differences":

$$S_i = S_0 \boxplus \nabla S_i$$
$$Y_i = S_i[0{:}64] \oplus S_i[64{:}128].$$

The Y_i and ∇S_i are known, $\nabla S_0 = 0$, and the problem consists in recovering S_0. We could probably encode it as an instance of SAT, feed it to a SAT-solver and be done with it.

Nevertheless, here is a detailed recovery procedure which obtain all bits of S_0, from right to left, by exploiting the non-linearity of modular addition. It takes negligible time. Let $\mathcal{C}_i$ the vector of (incoming) carries generated during the addition of S_0 and ∇S_i:

$$S_i[j] = S_0[j] \oplus \nabla S_i[j] \oplus \mathcal{C}_i[j]$$
$$\mathcal{C}_i[j] = \begin{cases} 0 & \text{if } j = 0 \\ \text{MAJ}(S_0[j-1], \nabla S_i[j-1], \mathcal{C}_i[j-1]) & \text{if } j > 0 \end{cases}$$

Combining all the above, we have:

$$Y_i[j] = Y_0[j] \oplus \big(\nabla S_i[j] \oplus \nabla S_i[64+j]\big) \oplus \big(\mathcal{C}_i[j] \oplus \mathcal{C}_i[64+j]\big) \tag{10}$$

This useful equation enables an induction process.

- When $j = 0$, the 0-th carries are zero, and therefore eq. (10) reveals the 64-th carries:

$$\mathcal{C}_i[64+j] = \big(Y_0[j] \oplus Y_i[j]\big) \oplus \big(\nabla S_i[j] \oplus \nabla S_i[64+j]\big).$$

- Next, suppose that $\mathcal{C}_i[0{:}j]$, $S_0[0{:}j-1]$, $\mathcal{C}_i[64{:}64+j]$ and $S_0[64{:}64+j-1]$ are known, for all i. We can compute $\mathcal{C}_i[j] \oplus \mathcal{C}_i[64+j]$ for any i using eq. (10). We then look a a specific index $\mathtt{i} > 0$ such that

$$\nabla S_{\mathtt{i}}[j-1] \neq \mathcal{C}_{\mathtt{i}}[j-1] \quad \text{and} \quad \nabla S_{\mathtt{i}}[64+j-1] = \mathcal{C}_{\mathtt{i}}[64+j-1].$$

  The point is that, thanks to the majority functions, $\mathcal{C}_{\mathtt{i}}[j] = S_0[j-1]$ and $\mathcal{C}_{\mathtt{i}}[64+j] = \nabla S_{\mathtt{i}}[64+j-1]$. It follows that:

$$S_0[j-1] = Y_0[j-1] \oplus Y_{\mathtt{i}}[j-1] \oplus \big(\nabla S_{\mathtt{i}}[j-1] \oplus \nabla S_{\mathtt{i}}[64+j-1] \oplus \nabla S_{\mathtt{i}}[64+j-1]\big)$$

  From there, we also have $S_0[64+j-1] = Y_0[64+j-1] \oplus S_0[j-1]$, and the j-th carry bits can be computed normally.

The whole procedure is shown in algorithm 4. Note that once S_0 has been found, then all subsequent states can be computed with error using $S_i = S_0 \boxplus \nabla S_i$. In particular, computing S_1 gives c by $c \leftarrow S_1 \boxminus aS_0$. This complete the reconstruction procedure for PCG64.

6 Implementation and Practical Results

We have implemented the state reconstruction algorithms described above using a mixture of C (for the computationally expensive parts) and Python (for the rest). We used the `fplll` library [dt16] to solve CVP instances exactly in dimension 63.

In this section, we briefly outline important aspects of our implementations and present practical results. Our codes are available in the supplementary material as well as online at:

Algorithm 4 Full state reconstruction algorithm

function ReconstructState($\Delta S_0, r_0, \ldots, r_k, X_0, \ldots, X_k$)
  for $i = 0, 1, \ldots, k$ **do** ▷ Setup
    $\nabla S_i \leftarrow \Delta S_0 (a^i - 1)(a-1)^{-1} \bmod 2^{128}$
    $Y_i \leftarrow X_i \lll r_i$ ▷ Undo rotations
    $\mathcal{C}_i[0] \leftarrow 0$ ▷ Bootstrap induction
    $\mathcal{C}_i[64] \leftarrow \big(Y_i[0] \oplus Y_i[j]\big) \oplus \big(\nabla S_i[j] \oplus \nabla S_i[64+j]$
  for $j = 1, 2, \ldots, 64$ **do** ▷ Induction
    $\mathfrak{i} \leftarrow \perp$ ▷ Find good index
    for $k = 1, 2, \ldots, k$ **do**
      if $\nabla S_k[j-1] \neq \mathcal{C}_k[j-1] \wedge \nabla S_k[64+j-1] = \mathcal{C}_k[64+j-1]$ **then**
        $\mathfrak{i} \leftarrow k$
    if $\mathfrak{i} = \perp$ **then** ▷ No suitable indice found?
      Abort with Failure
    ▷ Compute next state bit
    $S_0[j-1] \leftarrow Y_0[j-1] \oplus Y_{\mathfrak{i}}[j-1] \oplus \big(\nabla S_{\mathfrak{i}}[j-1] \oplus \nabla S_{\mathfrak{i}}[64+j-1] \oplus \nabla S_{\mathfrak{i}}[64+j-1]\big)$
    $S_0[64+j-1] \leftarrow Y_0[64+j-1] \oplus S_0[j-1]$
    for $i = 0, 1, \ldots, k$ **do** ▷ Compute next carries
      $\mathcal{C}_i[j] \leftarrow \mathrm{MAJ}(S_0[j-1], \nabla S_i[j-1], \mathcal{C}_i[j-1])$
      $\mathcal{C}_i[64+j] \leftarrow \mathrm{MAJ}(S_0[64+j-1], \nabla S_i[64+j-1], \mathcal{C}_i[64+j-1])$
  return S_0

Table 2: Empirical success probabilities with smaller parameters.

$n = 3$ (section 4)	
ℓ	Success proba.
16	≈ 0.125
17	≈ 0.25
18	≈ 0.5
19	1
20	1 (proved)

$n = 4$ (section 5.1)	
ℓ	Success proba.
10	≈ 0.12
11	≈ 0.64
12	≈ 0.995
13	1
14	1 (proved)

https://github.com/cbouilla/pcg/

The designer of PCG was kind enough to send us two sets challenge inputs: one with the default (known) increment and one with a random secret increment. She generated random seeds and provided us with the first outputs of the pseudo-random generator. We were able to reconstruct the seed with an extremely high confidence level, because they re-generate the same outputs. We emailed back the seeds and received confirmation that they were indeed correct.

We have therefore successfully taken the challenge of predicting the output of the PCG64 generator.

The analysis of section 3 yields parameters that guarantee that the reconstruction procedure *always* succeeds. In most cases, these parameters are pessimistic. We ran a series of experiments to determine more practical choices: using smaller-than-guaranteed values of ℓ (the number of guessed least-significant bits), we measured the success probability of the state reconstruction procedure. The results are shown in table 2.

6.1 Known Increment

When the increment c is known, algorithm 1 is all it takes to reconstruct the internal state of the generator and predict it (or output the seed). We implemented it in C, using

OpenMP to parallelize the outer loop that guess the least-significant bits of the state. This yields a simple multi-core implementation. We used the `gcc 8.3.0` compiler.

From section 3.3, we know that guessing $\ell = 20$ least-significant bits ensures deterministic success. However, we observed empirically that $\ell = 19$ works with probability ≈ 1, and runs twice as fast. $\ell = 18$ and $\ell = 17$ run with probability $\approx 1/2$ and $\approx 1/4$ respectively, therefore are much less useful. In practice, we used $\ell = 19$.

We ran it on a server equipped with two 16-core Intel Xeon Gold 6130 CPU @ 2.10GHz ("*Skylake*") CPUs. The inner loop does 2^{37} iterations and terminates in 42.3s, which makes 23 core minutes.

These processors operate at a different frequency depending on the number of cores used and the type of instructions executed. Our code uses only scalar instructions, so the CPUs runs at the highest frequency tier when executing it. Using a single software thread per physical core (each core presents two hardware execution contexts, commercially called *HyperThreads*) allows the CPU to run at 2.8Ghz, the maximum "Turbo" frequency on all cores. Using one software thread per hardware thread reduces the frequency to $\approx$ 2.6Ghz, but allows to better saturate the execution units of the CPU and yields a nearly 20% speedup overall.

Therefore the algorithm requires $2^{41.67}$ CPU cycles in total; this makes less than 5 cycles per iteration of the inner loop. We used several implementation tricks to reach this level of efficiency:

- We used the `__uint128_t` type provided by most C compilers to do 128-bit arithmetic when computing S_1 from S_0. Apart from that, the algorithm has been designed to do mostly 64-bit arithmetic, for the sake of efficiency.
- Looking at the algorithm, it is clear that U_1 and U_2 are actually not needed, so we just don't compute them.
- $\widetilde{T}_J$ is a function of w, j and r_J (with $j = 0, 1, 2$). therefore, for each new value of w, we precompute once and for all an array indexed by (J, R_J) of the 192 possible values of $\widetilde{T}_J$.
- Pushing the same idea a bit further, we precompute parts of the matrix-vector product inside the rounding: this computes a linear combination of the rows of G_3^{-1}, in which $\widetilde{T}_j$ is the coefficient of the j-th row. So we precompute the 576 possible products $\widetilde{T}_j \cdot G_3^{-1}[j, k]$.
- We enumerate the possible rotations in lexicographic order. This means that $\widetilde{T}_0$ changes in each iteration while $\widetilde{T}_1$ (resp $\widetilde{T}_2$) changes every 64 (resp 4096) iterations. Therefore, in 98% of the iterations, two-thirds of the matrix-vector product inside the rounding are the same as from the previous iteration. Therefore, we fully compute the matrix-vector product only when r_1 changes and only update it when r_0 changes.
- The rounding operation, when done naively by writing `llround(x)`, is actually a bottleneck: it calls a library function that accounted for about 20% of the total running time. We instead used the following technique, which correctly returns $\lfloor x \rceil$ whenever $|x| < 2^{51}$:

```
long long fast_round(double x)
{
    union { double d; long long l; } magic;
    magic.d = x + 6755399441055744.0;
    magic.l <<= 13;
    magic.l >>= 13;
```

```
    return magic.l;
}
```

This hack exploits the IEEE754 representation of double-precision floats: the mantissa lies in bits [0:52] while the sign bit and the exponents take the 12 most significant bits. Adding $2^{52} + 2^{51}$ forces the mantissa to shift to the correct position and inserts an extra 1 bit at position 51. The two shifts clear the extra bit and the exponent, while correctly expanding the sign bit.

6.2 Unknown Increment

When the increment c is known, the internal state of PCG64 can be practically reconstructed from $X_0, \ldots, X_{63}$ using the algorithms shown in section 5. Only algorithm 2 is computationally expensive; we implemented it in C, while we implemented algorithms 3 and 4 in Python.

We have shown that algorithm 2 is correct when $\ell = 14$. The procedure does $2^{29+2\ell}$ iterations of the inner loop, so decreasing ℓ would really be interesting. Looking at table 2, we settle for $\ell = 13$ in the worst case; let T denotes the running time when $\ell = 13$.

It seems that the most promising strategy consists in choosing $\ell = 11$; if the reconstruction procedure fails, then we try again with different inputs. The expected running time of this approach number of trials is $T/(16 \times 0.64) \approx T/10.25$. In our implementation, $T = 200,000$ CPU hours, so the expected running time of the reconstruction procedure is about $20,000$ CPU hours. In fact we were lucky: on the challenge input, the first attempt with $\ell = 11$ succeeded, so the whole process took only $12,500$ CPU hours.

It actually ran in 35 wall-clock minutes using 512 cluster nodes, each equipped with two 20-cores Intel Xeon Gold 6248 @ 2.5Ghz ("*Cascade Lake*"). The actual machine is the `jean-zay` computer located at the IDRIS national computation center. Note that on this particular parallel computer, running the algorithm with $\ell = 13$ would take 10 hours using the same amount of resources, so the whole procedure *is* practical, even in the absolute worst case.

The outer loop of algorithm 2 makes $2^{2\ell-1}$ iterations while the inner loop makes 2^{30} iterations. Using a single hardware execution context, we measured that one of the outer loop takes between 41.5s and 44s (apparently not all nodes of the cluster are running at exactly the same speed, potentially because of "turbo boost" adjustments and thermal constraints). Because of this variability, we implemented a master-slave work distribution pattern, in which a master process dispatches iterations of the outer loop to slave processes. This also made checkpointing very easy. We used MPI for inter-process communication.

With $\ell = 11$, the whole process took $2^{56.74}$ CPU cycles, which makes less than 6 cycles per iteration of the inner loop. We used essentially the same implementation tricks discussed above. However, this time we had to additionally implement the ConsistencyCheck procedure, which is called in the inner loop. We observed that the set of possible candidate values $\mathcal{C}$ only depends on w_0 (the variable of the outer loop). Therefore, before entering the inner loop, we precompute a bit field of size 2^ℓ describing $\mathcal{C}_i$. To simplify the implementation, we flatten them by computing $\mathcal{C} = \cup_i \mathcal{C}_i$. This slightly increase the probability of false positives, but makes our code slightly simpler.

7 Conclusion

We have presented a practical state reconstruction algorithm for the PCG64 conventional pseudo-random number generator, the default in the `NumPy` library. In the worst case, we recovers all the secret information using 512 consecutive output bytes, using 2.3 CPU years of computation. We have executed the algorithm in practice using a large parallel computer. The PCG64 generator is fast and not intended for cryptographic purposes; we

have shown that, in practice, this comes at the price of strong pseudo-randomness. It should absolutely not be used when unpredictability of the random numbers is required, for fear of practical attacks.

On the other hand, our results do not mean that PCG64 should be deprecated for scientific computing. But they do mean that its output has detectable properties. Whether these properties may affect the results of Monte-Carlo numerical simulations is another matter entirely.

Acknowledgements. The authors thank Thomas Espitau, Damien Vergnaud and Vincent Zucca for helpful discussions. We also thank Léo Ducas for helping us with the G6K library and Melissa O'Neil for kindly indulging our request for challenge inputs.

This work was granted access to the HPC resources of IDRIS under the allocation 2019-A0060610749 made by GENCI.

References

[ADH+19] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019.

[Bab86] László Babai. On lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[Boy89] Joan Boyar. Inferring sequences produced by a linear congruential generator missing low-order bits. *Journal of Cryptology*, 1(3):177–184, Oct 1989.

[dt16] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016.

[FHK+88] Alan M. Frieze, Johan Hastad, Ravi Kannan, Jeffrey C. Lagarias, and Adi Shamir. Reconstructing truncated integer variables satisfying linear congruences. *SIAM J. Comput.*, 17(2):262–280, April 1988.

[FLW92] Alan M. Ferrenberg, D. P. Landau, and Y. Joanna Wong. Monte carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.*, 69:3382–3384, Dec 1992.

[JS98] Antoine Joux and Jacques Stern. Lattice reduction: A toolbox for the cryptanalyst. *J. Cryptology*, 11(3):161–185, 1998.

[Knu85] D. Knuth. Deciphering a linear congruential encryption. *IEEE Transactions on Information Theory*, 31(1):49–52, 1985.

[Knu98] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1998.

[Leh49] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*, volume 26 of *Annals of the Computation Laboratory of Harvard University*, pages 141–146, 1949. Available online at http://www.bitsavers.org/

pdf/harvard/Proceedings_of_a_Second_Symposium_on_Large-Scale_Digital_Calculating_Machinery_Sep49.pdf.

[Lem19] Daniel Lemire. The fastest conventional random number generator that can pass Big Crush?, 2019. Blog entry. Last accessed [31-05-2020]. Available online at https://lemire.me/blog/2019/03/19/the-fastest-conventional-random-number-generator-that-can-pass-big-crush/.

[LS07] Pierre L'Ecuyer and Richard J. Simard. Testu01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, 2007.

[Mar03] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[O'N14a] Melissa. E. O'Neill. Pcg, a family of better random number generators, 2014. http://www.pcg-random.org/.

[O'N14b] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014. Available online at https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf.

[Riv94] Ronald L. Rivest. The RC5 encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer, 1994.

[SMDS11] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[vCV11] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

[Vig17] Sebastiano Vigna. Further scramblings of marsaglia's xorshift generators. *J. Comput. Appl. Math.*, 315:175–181, 2017.

[vN51] John von Neumann. Various techniques used in connection with random digits. In A. S. Householder, G. E. Forsythe, and H. H. Germond, editors, *Monte Carlo Method*, volume 12 of *National Bureau of Standards Applied Mathematics Series*, chapter 13, pages 36–38. US Government Printing Office, Washington, DC, 1951. Available online at https://mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf.

[YRS+09] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 debian openssl vulnerability. In Anja Feldmann and Laurent Mathy, editors, *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009*, pages 15–27. ACM, 2009.

[ZML+16] Shuangyi Zhu, Yuan Ma, Jingqiang Lin, Jia Zhuang, and Jiwu Jing. More powerful and reliable second-level statistical randomness tests for NIST SP 800-22. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 307–329, 2016.