Practical Seed-Recovery for the PCG Pseudo-Random Number Generator

Charles Bouillaguet, Florette Martinez and Julia Sauvage

November 2, 2020

Introduction

What?

Cryptanalysis of the **Permuted Congruential Generator** (PCG).

Why?

https://www.pcg-random.org

PCG, A Better Random Number Generator | Download | Docs | Paper | Video | Blog

PCG, A Family of Better Random Number Generators

PCG is a family of simple fast space-efficient statistically good algorithms for random number generation. Unlike many general-purpose RNGs, they are also hard to predict.

At-a-Glance Summary

	Statistical Quality	Prediction Difficulty	Reproducible Results	Multiple Streams	Period	Useful Features	Time Perfomance	Space Usage	Code Size & Complexity	k-Dimensional Equidistribution
PCG Family	Excellent	Challenging	Yes	Yes (e.g. 2^{63})	Arbitrary	Jump ahead, Distance	Very fast	Very compact	Very small	Arbitrary*
Mersenne Twister	Some Failures	Easy	Yes	No	Huge 2^{19937}	Jump ahead	Acceptable	Huge (2 KB)	Complex	623
Arc4Random	Some Issues	Secure	Not Always	No	Huge 2^{1699}	No	Slow	Large (0.5 KB)	Complex	No
ChaCha20†	Good	Secure	Yes	Yes (2^{128})	2^{128}	Jump ahead, Distance	Fairly Slow	Plump (0.1 KB)	Complex	No

Why?

G, A Family of Better Random Number G

PCG is a family of simple fast space-efficient statistically good algorit
Unlike many general-purpose RNGs, they are also hard to predict.

Why?

At-a-Glance Summary

	Statistical Quality	Prediction Difficulty	Reproducible Results	Multiple Streams	Period	
PCG Family	Excellent	Challenging	Yes	Yes (e.g. 2^{63})	Arbitrary	J ah Dis
Mersenne	Some	Easy	Yes	No	Huge	J


CHALLENGE ACCEPTED

Introduction

What?

Cryptanalysis of the **Permuted Congruential Generator** (PCG).

Results

Practical seed-recovery / prediction.

How?

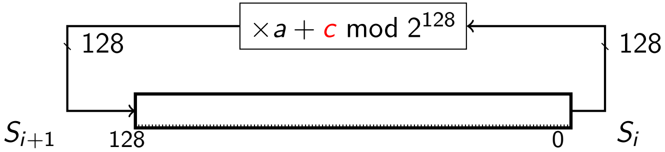
- "Guess-and-Determine" attack.
- Most expensive part : many small CVP problems.
- **Actually done** in $\leq$ 20 000 CPU-hours.

Permuted Congruencial Generators (PCG)

- Conventional (non-crypto) pseudo-random generators
- Designed in 2014 by Melissa O'Neil
- PCG64
  - Internal state : 128-bit state and 128-bit increment
  - 64-bit outputs
  - 256-bit seed (or 128-bit with default increment)
  - Default pseudo-random generator in NumPy

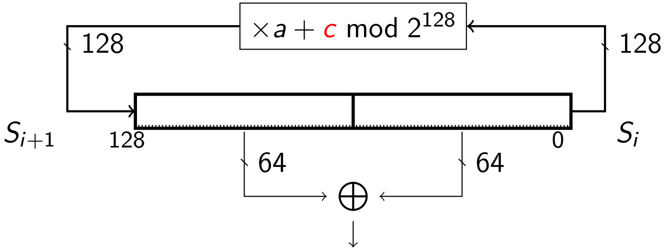
Permuted Congruencial Generators (PCG)

- Conventional (non-crypto) pseudo-random generators
- Designed in 2014 by Melissa O'Neil
- PCG64
  - Internal state : 128-bit state and 128-bit increment
  - 64-bit outputs
  - 256-bit seed (or 128-bit with default increment)
  - Default pseudo-random generator in NumPy



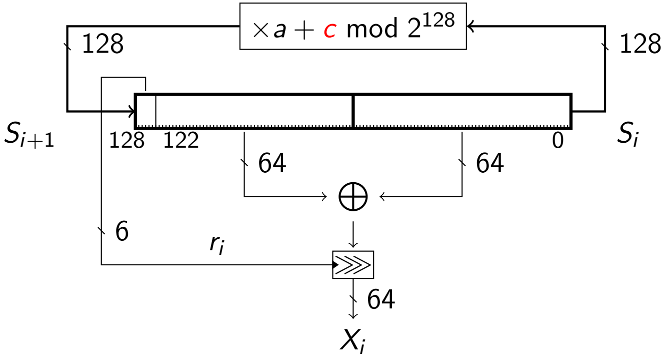
Permuted Congruencial Generators (PCG)

- Conventional (non-crypto) pseudo-random generators
- Designed in 2014 by Melissa O'Neil
- PCG64
  - Internal state : 128-bit state and 128-bit increment
  - 64-bit outputs
  - 256-bit seed (or 128-bit with default increment)
  - Default pseudo-random generator in NumPy



Permuted Congruencial Generators (PCG)

- Conventional (non-crypto) pseudo-random generators
- Designed in 2014 by Melissa O'Neil
- PCG64
  - Internal state : 128-bit state and 128-bit increment
  - 64-bit outputs
  - 256-bit seed (or 128-bit with default increment)
  - Default pseudo-random generator in NumPy



Attack Outline

- **Guess** some bits in a few successive states.
  - Least-significant bits
  - Rotations

$\Rightarrow$ Turn it into a **(regular) truncated congruential generator**.

- **Reconstruct** hidden information using lattice techniques.

- **Discard** bad guesses.

Attack Outline

- **Guess** some bits in a few successive states.
  - Least-significant bits
  - Rotations

$\Rightarrow$ Turn it into a **(regular) truncated congruential generator**.

- **Reconstruct** hidden information using lattice techniques.
  - Easy case (c known): full state
  - Hard case (c unknown): only partial information

- **Discard** bad guesses.

Easy Case: Known increment

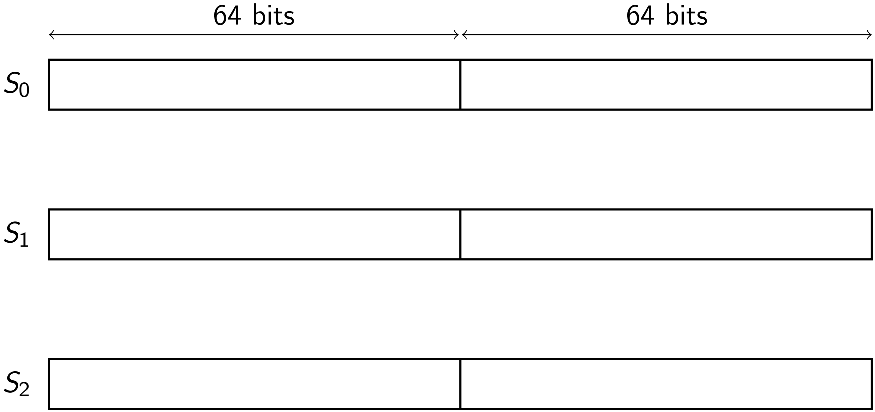
If the increment (c) is **known**...

... Get rid of it!

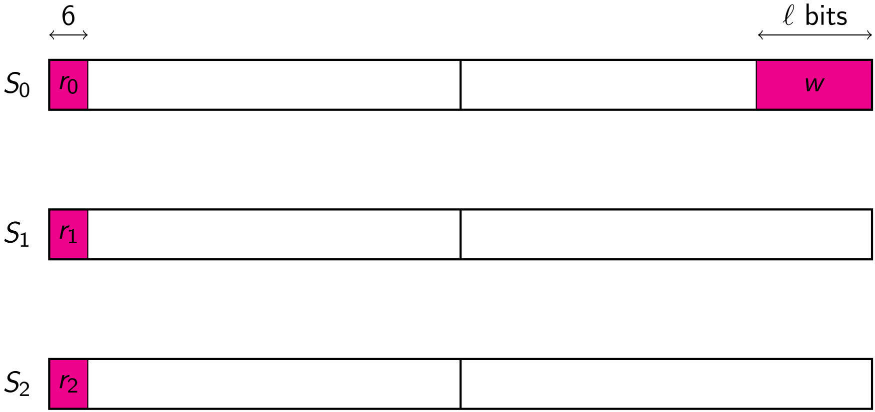
- $S'_0 \leftarrow S_0$
- $S'_1 \leftarrow S_1 - c$
- $S'_2 \leftarrow S_2 - (a+1)c$
- $S'_3 \leftarrow S_3 - (a^2+a+1)c$
- $\vdots$

Yields S' : sequence of states with $c = 0$
→ **Geometric sequence**.

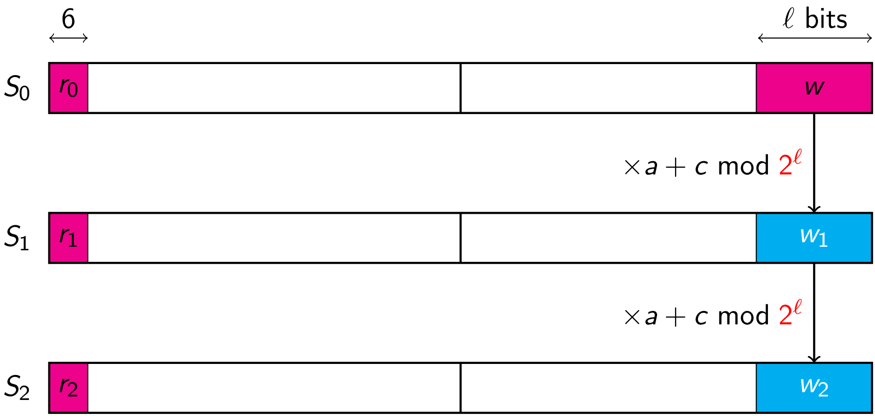
Attack Details



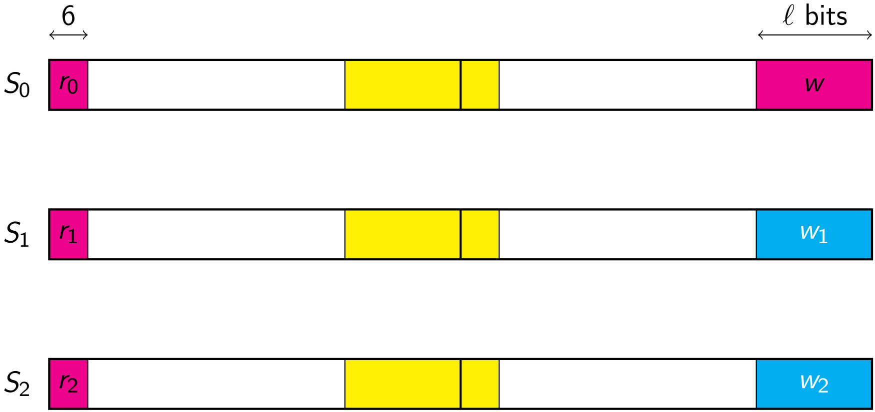
Attack Details



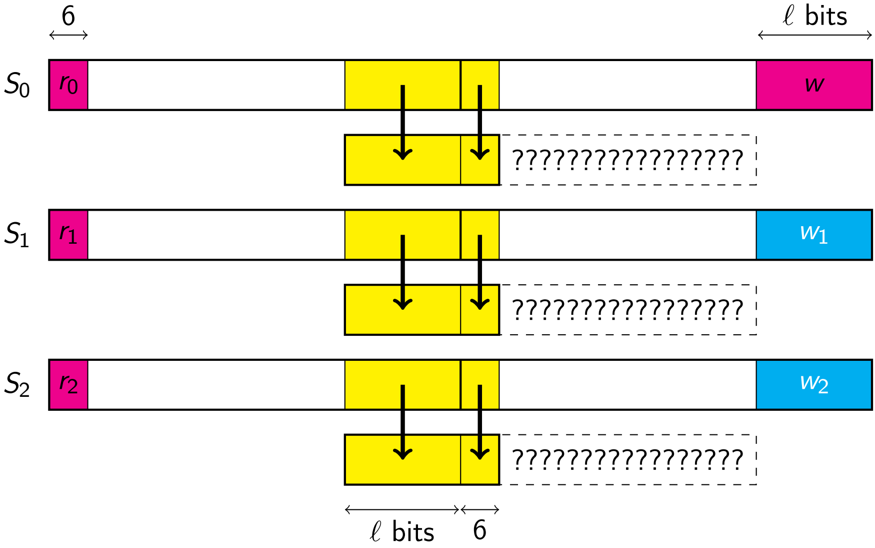
Attack Details



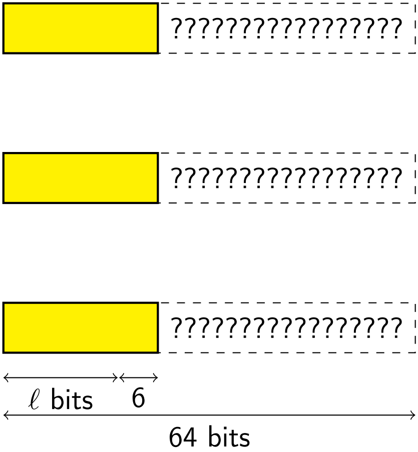
Attack Details



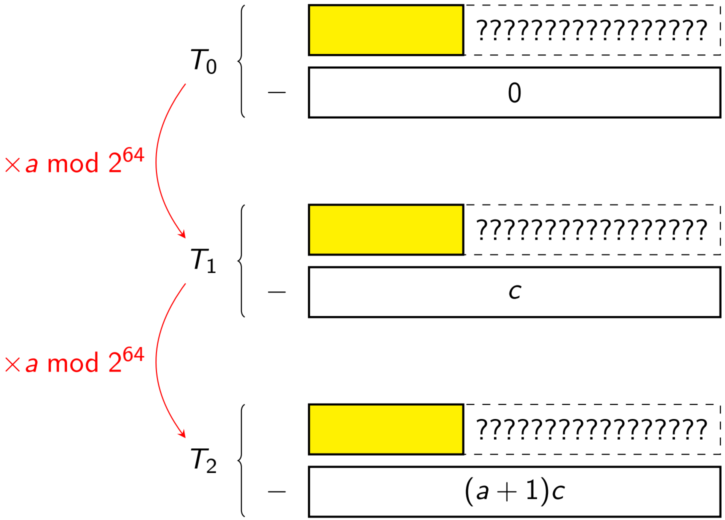
Attack Details



Attack Details

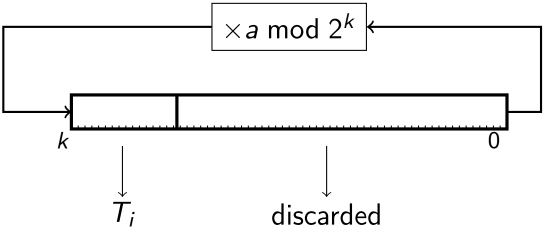


Remove the "Constant Component"



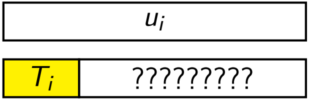
Truncated Linear Congruential Generators

- Internal state : 2^k-bit state.
- Multiplier a: known constant.
- Initial state: unknown 2^k-bit seed.



Reconstructing Truncated Geometric Sequences

- Sequence $u_{i+1} = a \times u_i \bmod 2^k$.
- T = Truncated version (low-order bits unknown).
- $\mathcal{L}$ = lattice spawned by the rows of



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

Main Idea

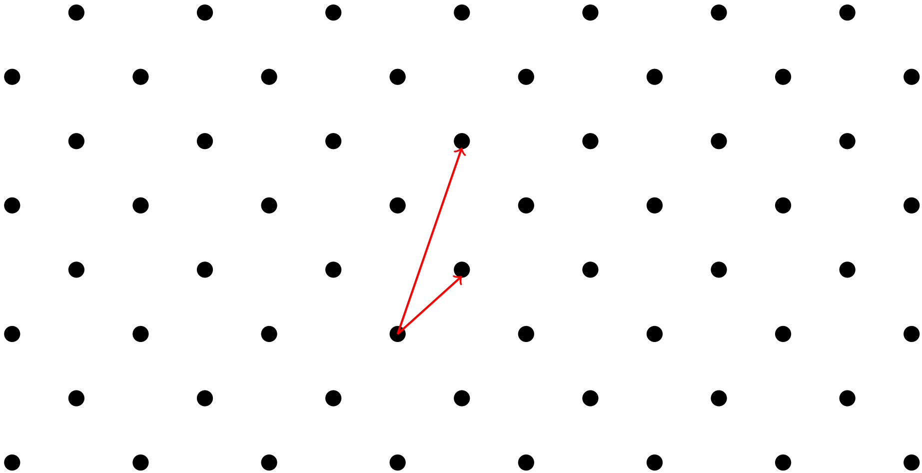
- $\mathbf{u} = (u_0, u_1, \dots, u_{n-1})$ **belongs** to the lattice $\mathcal{L}$.
- T (truncated geometric series) is an **approximation** of $\mathbf{u}$.

$\Rightarrow$ T is **close** to a point of $\mathcal{L}$.

$\Rightarrow$ **Closest** point to T in $\mathcal{L} \rightsquigarrow \mathbf{u}$.

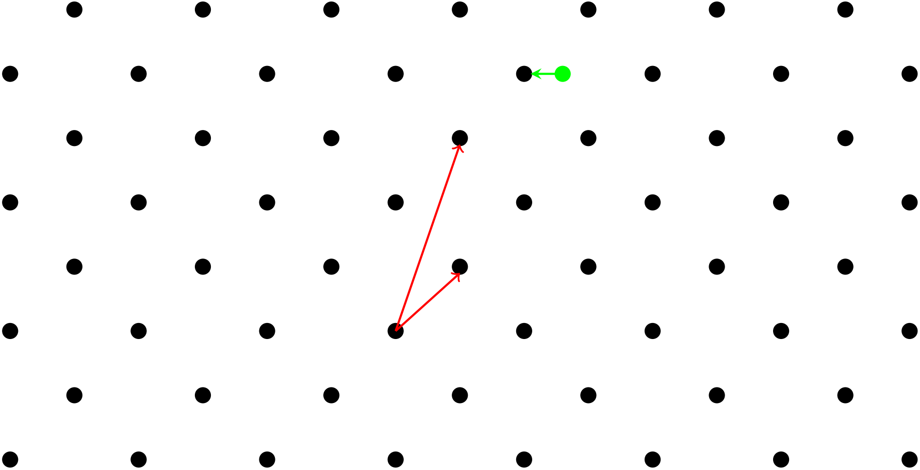
Lattices and Basis reduction

- Lattice : subgroup of $\mathbb{R}^n$ isomorphic to $\mathbb{Z}^m$



Lattices and Basis reduction

- Lattice : subgroup of $\mathbb{R}^n$ isomorphic to $\mathbb{Z}^m$



CVP problem and Babai rounding

Closest Vector Problem

- Standard **NP-hard** problem on lattices.
- Given arbitrary $\mathbf{x} \in \mathbb{Z}^n$, find closest lattice point.

Babai Rounding Algorithm

- Approximately solves CVP.

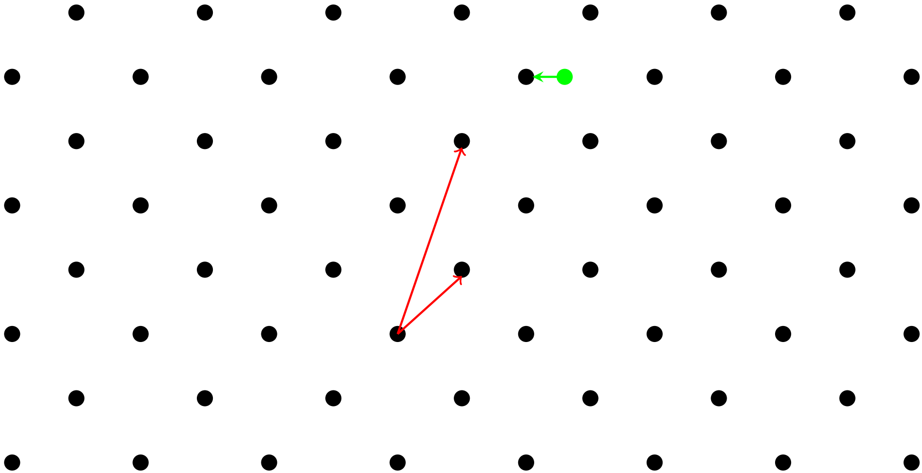
$$\mathit{BabaiRounding}(\mathbf{x}, \mathcal{L}) = H \times \mathrm{round}\left(H^{-1} \times \mathbf{x}\right)$$

  Where H is a "good" (LLL-reduced) basis of the lattice $\mathcal{L}$.
- FAST (two matrix-vector products + rounding)
- Exponentially bad approximation (in the lattice dimension).

$\rightarrow$ Often exact in small dimension though.

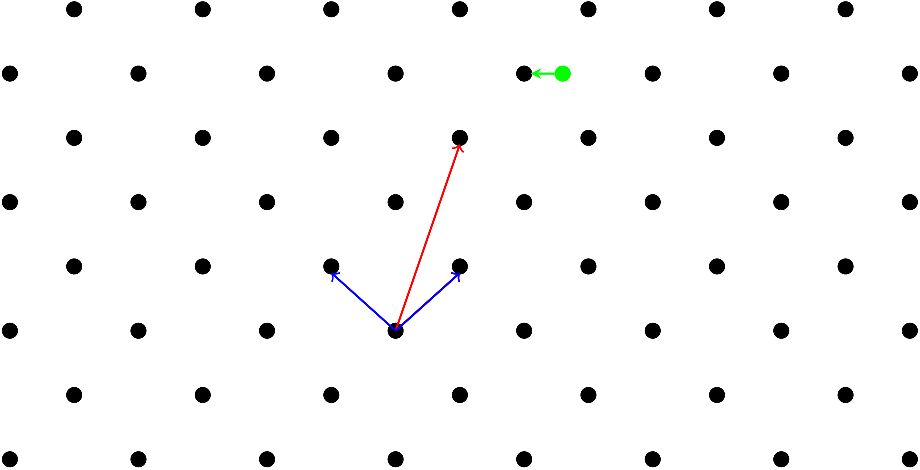
Lattices and Basis reduction

- Lattice : subgroup of $\mathbb{R}^n$ isomorphic to $\mathbb{Z}^m$



Lattices and Basis reduction

- Lattice : subgroup of $\mathbb{R}^n$ isomorphic to $\mathbb{Z}^m$



Implementation (Easy case, known increment)

Summary

- Observe 3 outputs X_0, X_1, X_2 (192 bits).
- Guess 37 bits:
  - $n = 3$ successive rotations (6 bits each),
  - $\ell = 19$ least significant bits of S_0,
- Solve 2^{37} instances of CVP in dimension 3 (Babai Rounding).
- Reconstruct initial state, check outputs.

Caveat

Attack proved correct for $\ell = 20$, works fine for $\ell = 19$...

Concretely...

- 25 CPU cycles per guess, 23 CPU-minutes in total.

Issue with c unknown

Summary so far (the **Easy Case**)

- The increment (c) is **known**:
  - Remove it, get truncated geometric sequence, CVP.

Now the **Hard Case**

- The increment (c) is **unknown**:
  - How to get truncated geometric sequence?
  - Use $\Delta S_i = S_{i+1} - S_i$ $(\Delta S_{i+1} = a \times \Delta S_i \bmod 2^{128})$.

Issue with c unknown

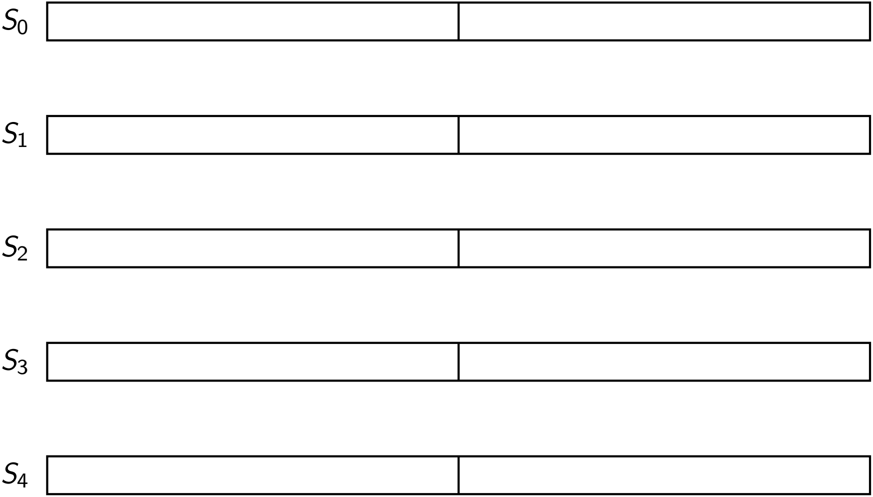
Summary so far (the **Easy Case**)

- The increment (c) is **known**:
  - Remove it, get truncated geometric sequence, CVP.

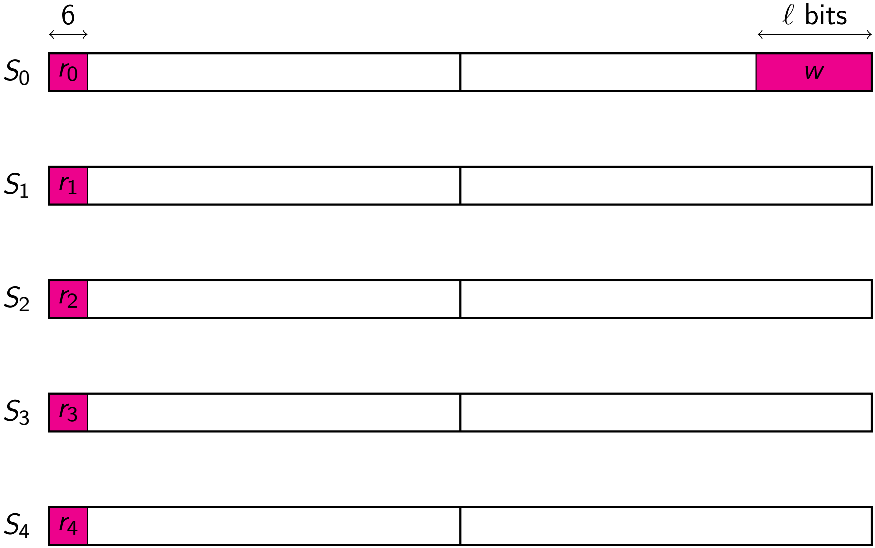
Now the **Hard Case**

- The increment (c) is **unknown**:
  - How to get truncated geometric sequence?
  - Use $\Delta S_i = S_{i+1} - S_i$ $(\Delta S_{i+1} = a \times \Delta S_i \bmod 2^{128})$.
- Same attack as before, but...
  - Must guess one more rotation.
  - Must guess least-significant bits of c.

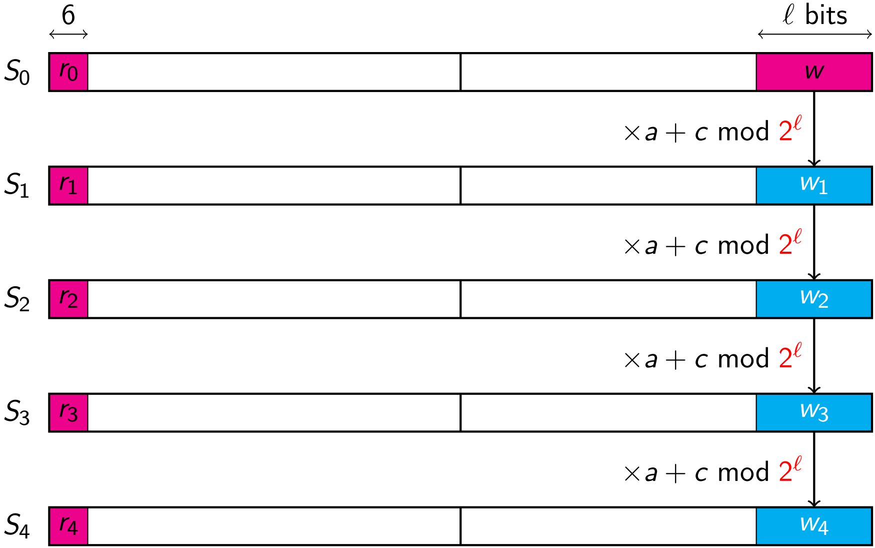
Attack Details



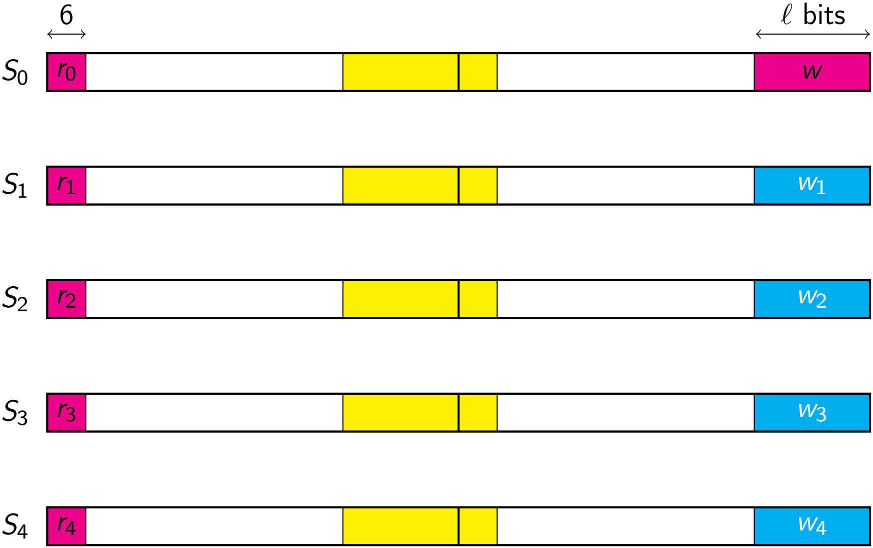
Attack Details



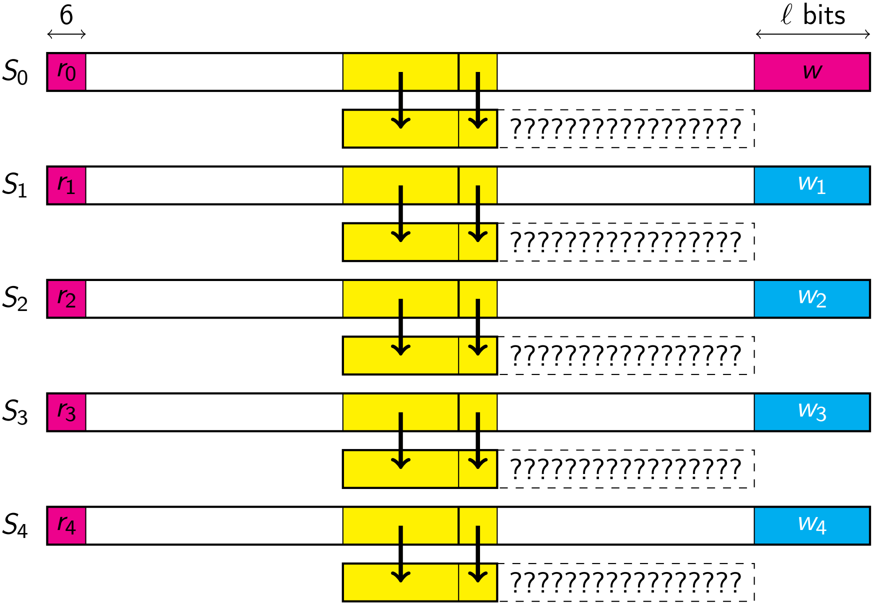
Attack Details



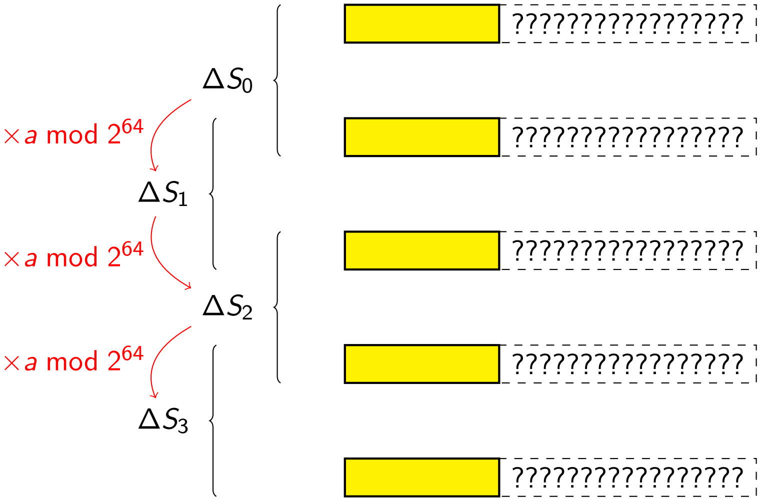
Attack Details



Attack Details



Attack Details



Attack Details (cont'd)

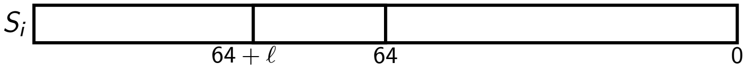
Summary so far

- **Guess** parts of the states (S_i).
- Attack state **differences** (ΔS_i).
- CVP in dim. 4 $\rightsquigarrow$ reconstruct partial ΔS_i (for all i).

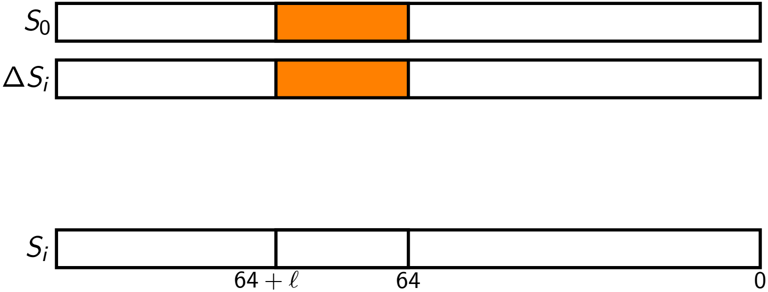
Problem

How to check if guesses are valid?

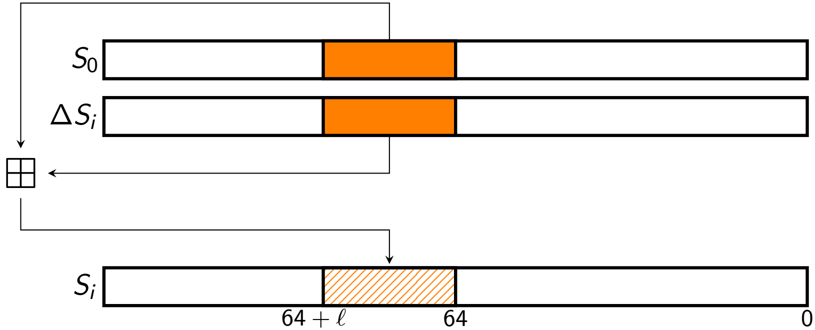
Consistency Check



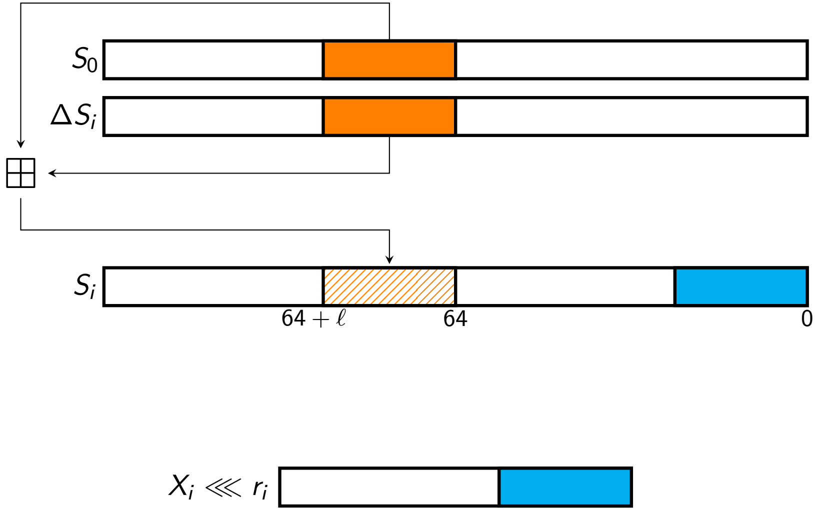
Consistency Check



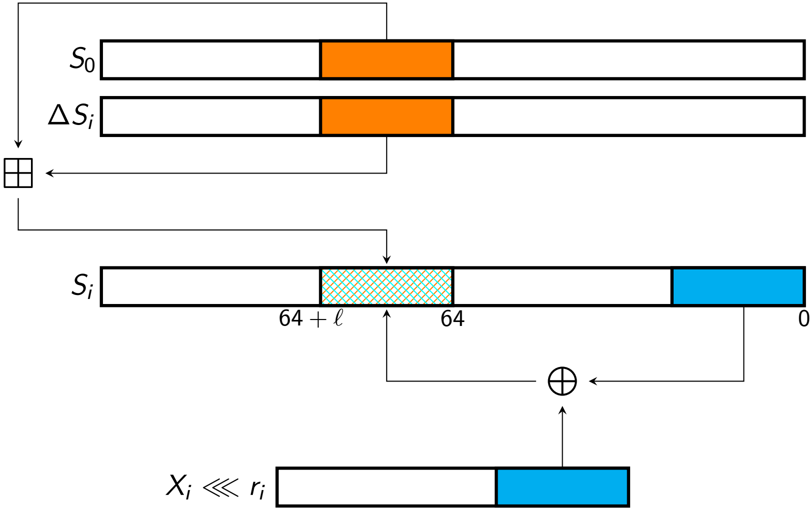
Consistency Check



Consistency Check



Consistency Check



Attack Details (cont'd)

Summary so far

- **Guess** parts of the states (S_i).
- Attack state **differences** (ΔS_i).
- CVP in dim. 4 $\rightsquigarrow$ reconstruct partial ΔS_i (for all i).

Problem

How to check if guesses are valid?

Solution

- $S_i[64 : 64 + \ell]$ from guesses + X_i (output) + r_i (rotation).
- $S_i[64 : 64 + \ell]$ from guesses + partial ΔS_i.

$\Rightarrow$ Try all possible r_i's. No match $\rightsquigarrow$ bad guess.

Finishing it Off

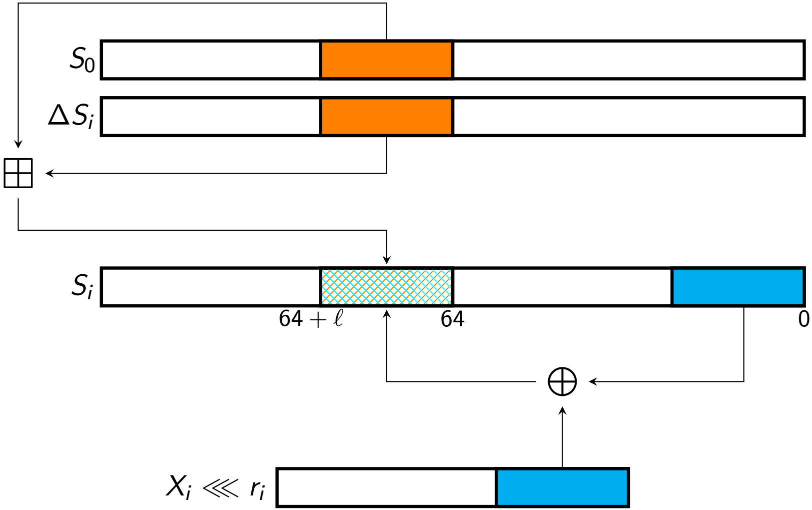
Summary so far

- **Guessed** parts of the states (S_i).
- Isolated **correct** guess $\rightsquigarrow$ correct partial differences ΔS_i.

Problem

How to get full initial state S_0?

Consistency Check



Finishing it Off

Summary so far

- **Guessed** parts of the states (S_i).
- Isolated **correct** guess $\rightsquigarrow$ correct partial differences ΔS_i.

Problem

How to get full initial state S_0?

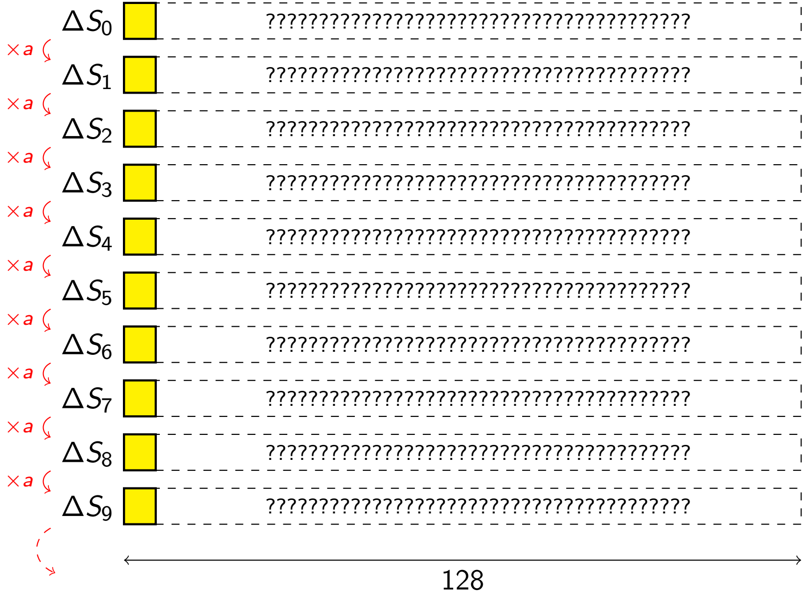
Solution

- Correct partial ΔS_i + consistency check $\rightsquigarrow$ **all** rotations r_i.

$\Rightarrow$ MSB of all S_i $\rightsquigarrow$ MSB of all ΔS_i.

$\Rightarrow$ CVP in dim. 64 $\rightsquigarrow$ full ΔS_0.

Reconstructing the Full Differences (CVP in dim. 64)



Finishing it Off

Summary so far

- **Guessed** parts of the states (S_i).
- Isolated **correct** guess $\rightsquigarrow$ correct partial differences ΔS_i.

Problem

How to get full initial state S_0?

Solution

- Correct partial ΔS_i + consistency check $\rightsquigarrow$ **all** rotations r_i.

$\Rightarrow$ MSB of all S_i $\rightsquigarrow$ MSB of all ΔS_i.

$\Rightarrow$ CVP in dim. 64 $\rightsquigarrow$ full ΔS_0.

- The rest is easy.

Implementation (Hard case, unknown increment)

Summary

- Observe 64 outputs (4096 bits).
- Guess k =51–55 bits:
  - $n = 5$ successive rotations (6 bits each),
  - $\ell = 11$–13 least significant bits of S_0 **and** c.
- Solve 2^k instances of CVP in dimension 4 (Babai Rounding).
- Consistency Check.

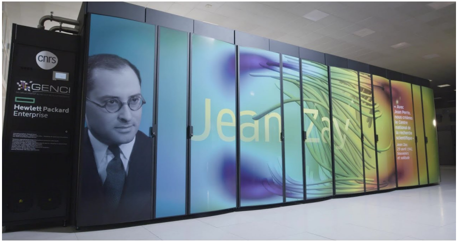
Caveat

- Attack proved correct for $\ell = 14$ (works fine for $\ell = 13$).
- Succeeds with $p = 0.66$ with $\ell = 11$.

Concretely...

- 55 CPU cycles per guess, 12.5k–20k CPU-hours in total.

Doing it for Real



- Used 512 nodes
  - 2×20-core Xeon Gold 6248 @ 2.5Ghz
- Running time: 35 minutes.

Conclusion

- Reconstructing the seed for PCG is **practical**.
- PCG is **not** cryptographically secure (never claimed to be).
- **Don't** use Numpy to generate nonces...