University of West Bohemia in Pilsen
Faculty of Applied Science

Semestral work:

SeedSeeker

Authors: Pavel Altmann, Jakub Kupčik,
Matěj Bartička, David Wimmer, Patrik Holub
Date: 1st July 2025

Contents

1 Description

In programming, generating random values is a common requirement, particularly for applications such as cryptography and systems requiring unpredictability (e.g., slot machines). If an attacker could predict these values, they could exploit this knowledge for malicious purposes.

Most systems employ Pseudo-Random Number Generators (PRNGs), which produce sequences of values through mathematical algorithms rather than true randomness. These algorithms generate sequences that satisfy specific statistical distributions (typically uniform distribution) and share a crucial characteristic: each value depends on previous values. Consequently, the entire sequence derives from an initial starting point called the seed.

Knowledge of the seed enables complete sequence prediction. While methods exist to deduce seeds for certain algorithms, no comprehensive tool previously implemented these techniques. Our solution addresses this gap.

We developed SeedSeeker, a Python application that attempts to reverse-engineer both the generator algorithm and its seed from output sequences. The implementation adopts a modular library approach, facilitating the addition of new generators and reversal methods as they become available.

2 User Manual

2.1 Download

SeedSeeker is available at GitHub (https://github.com/wren-projects/SeedSeeker). Users have two installation options:

- Download pre-built binaries from the Releases section (recommended for users without Python experience)
- Clone the repository and run directly via Python (requires dependency installation)

For Python execution, required dependencies can be installed manually or via tools like uv or Poetry.

2.2 Running

The tool's help page (accessible via command-line flags) and manpage provide comprehensive usage instructions. SeedSeeker operates in three primary modes:

1. **Generation**: Execute a specified generator with a given seed (see Table 2.1 for seed formats)
2. **Reversal**: Attempt to deduce the initial state from output sequences using all applicable reversal algorithms
3. **Prediction**: Generate future values from a known initial state

Important Limitations:

- Reversal algorithms perform optimally with standard generator implementations
- Degenerate states (e.g., constant output sequences) may produce incorrect predictions
- Successful reversal doesn't guarantee accurate future predictions for all generators

Table 2.1: Seed formats for generators

Generator	**Seed Format**	**Conditions**
LCG	"$m; a; c; x_0$"	$m > 0 \wedge$ $0 < a < m \wedge$ $0 \leq c \leq m \wedge$ $0 \leq x_0 < m$
Lagged Fibonacci	"$r; s; m; seed$" or "$r; s; m; s_0, s_1, ...$"	$m > 0 \wedge$ $0 < r < m \wedge$ $0 < s < m \wedge$ $r \neq s \wedge$ $seed > 0 \vee$ $s_0, s_1, ... \geq 0 \wedge$ $len(s_i) = max(r, s)$
ran3	"$seed$"	$seed \in \mathbb{R}$
Xoshiro256**	"$a; b; c; d$"	$a, b, c, d \geq 0 \wedge$ $0 < a, b, c, d \leq 2^{64}$
Mersenne Twister	"$seed$"	$0 \leq seed \leq 2^{32}$

3 Reversal Algorithms

3.1 LCG

Let x_i be the sequence of LCG output. The sequence is given by the recursive formula:

$$x_{i+1} = (a \cdot x_i + c) \mod M$$

We first create a second sequence:

$$y_i = x_{i+1} - x_i$$

Note $y_{i+1} = a \cdot y_i \mod M$.

Let us denote

$$r_i = y_i \cdot y_{i+3} - y_{i+1} \cdot y_{i+2}$$

And note $r_i \equiv 0 \mod M$.

So to retrieve the value of modulo, we can just take an arbitrary number of r_is, calculate their GCD and estimate M based on that value.

Once we have modulo, we can just multiply r_{i+1} by the inverse of r_i in modulus M to retrieve a. Finally we trivially retrieve c.

3.2 ran3

The output sequence directly reveals the initial state. By observing 55 consecutive values, we completely determine the generator's state.

3.3 Lagged Fibonacci

Let x_i be the sequence of outputs. We know, that $x_i = ((x_{i-r} + x_{i-s}) \mod M) + K$, where K is one if $x_{i-1} > x_{i-r} + x_{i-s}$. We reverse this generator by brute force.

We try out all possible values of r and s (lower than some threshold). Then, for each of them, we calculate $x_i - x_{i-r} - x_{i-s}$. We know this is either $M - 1, M, M + 1$ or 0. By doing this multiple times, we pinpoint get the exact value of M.

3.4 Xoshiro256**

This generator uses a total of 256 bits of inner state broken up into 4 `uint_64` variables (stored in array 's').

For full description, see here.

Let us explore, how does the inner state change in one iteration:

$$[A, B, C, D] \rightarrow [A \oplus B \oplus D, A \oplus B \oplus C, A \oplus C \oplus (B << 17), \text{rotl}(B \oplus D)]$$

, where $\oplus$ denotes bitwise XOR.

Note that the output is always reversible into the second element of given state.

We can retrieve `s[1]` and `s[0]^s[1]^s[2]` from the first 2 outputs.

Let us denote t the state after one iteration. Note we know the value of `t[1]` and `t[2]`.

From 3rd output, we can reconstruct `t[0]`.

3.5 Mersenne Twister

We incorporated the randcrack library for Mersenne Twister reversal. Refer to the source repository for implementation details.

4 Library Description

SeedSeeker's architecture facilitates extensibility through a modular library system. Adding new functionality requires:

1. Implementing a `GeneratorState` class
2. Implementing a `Generator` class
3. Developing a reversal method
4. Updating the import registry

The LCG implementation serves as a reference example. Import configurations reside in the CLI directory.