MATHEMATICA, Tome 51(74), N° 2, 2009, pp. 163–171

COMBINING THE SOLITAIRE ENCRYPTION ALGORITHM WITH LAGGED FIBONACCI PSEUDORANDOM NUMBER GENERATORS

CHRISTIAN SĂCĂREA, CSABA SZÁNTÓ and ISTVÁN ŞUTEU SZÖLLŐSI

Abstract. We use a "byte" variant of Bruce Schneier's Solitaire Encryption Algorithm to produce the seed and weight system of a lagged Fibonacci pseudorandom number generator which generates a sequence of bytes. We analyze variants of the procedure above by testing them using some up to date randomness tests.

MSC 2000. 65C10.

Key words. Pseudorandom number generator, lagged Fibonacci generator, Solitaire algorithm.

1. PSEUDORANDOM NUMBER GENERATORS, LAGGED FIBONACCI GENERATORS AND SOLITAIRE

According to [2] a pseudorandom number generator (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called seed. PRNG's are used in cryptography for generating a "keystream" of a so called stream cipher.

Confidence in the fact that a PRNG generates numbers that are sufficiently "random" to suit the intended use is usually gained through careful mathematical analysis. This is especially important in case of PRNG's for cryptographic use. There is an up to date collection of 15 tests which can help in testing the quality of our generated sequence (mainly for cryptographic use). The implementation of these tests and further details can be found in [3].

Common classes of PRNG algorithms are linear congruential generators, lagged Fibonacci generators, linear feedback shift registers and generalized feedback shift registers. Recent instances of pseudo-random algorithms include Blum Blum Shub, Fortuna, and the Mersenne twister.

We will focus on lagged Fibonacci generators. These are defined by the following recurrence

$$X_i := (a_1 X_{i-1} + ... + a_r X_{i-r}) \mod m,$$

where $r > 1$ and $a_r \neq 0$. The seed in this case is the sequence of initial values $X_1, ..., X_r \in \mathbb{Z}_m$. Here $a_1, ..., a_r$ will be called weights.

The maximum length of a pseudorandom sequence before it begins to repeat is called period. Of course changing the seed will change the period and an

important goal is to have huge periods. A seed is weak if it produces a shorter period or if the generated sequence is failing certain statistical tests.

One can see that in case of a lagged Fibonacci generator the maximal period is $m^r - 1$. So with fixed weights a huge m and a big r will increase periods. In contrast, our aim is to increase the period and cryptographic quality of lagged Fibonacci generators not by increasing m or r but rather periodically changing the weights. In order to do that we use as a weight changing algorithm Bruce Schneier's Solitaire. More precisely we need to slightly modify the Solitaire algorithm such that it generates bytes instead of letters. To achieve this we keep the steps of the algorithm the only modification being the size of the "deck". Instead of a deck with 52 cards and two jokers we will use a virtual deck of $2^8 = 256$ cards (numbered from 0 to 255) and two jokers (A and B). An unkeyed deck will mean in this situation the virtual cards in increasing order (followed by the jokers A and B). We will use a key to shuffle this unkeyed deck converting it into a keyed deck. From now on Solitaire will mean for us this modified "byte" version of the initial algorithm which produces a sequence of bytes.

The steps of this algorithm are (as taken from [4])

- Key the unkeyed deck (see * after the last step) and use this as your initial deck.
- Find the A joker. Move it one card down. (That is, swap it with the card beneath it.) If the joker is the bottom card of the deck, move it just below the top card.
- Find the B joker. Move it two cards down. If the joker is the bottom card of the deck, move it just below the second card. If the joker is one up from the bottom card, move it just below the top card.
- Perform a triple cut. That is, swap the cards above the first joker with the cards below the second joker. "First" and "second" jokers refer to whatever joker is nearest to, and furthest from, the top of the deck. Ignore the "A" and "B" designations for this step. Remember that the jokers and the cards between them don't move; the other cards move around them. If there are no cards in one of the three sections (either the jokers are adjacent, or one is on top or the bottom), just treat that section as empty and move it anyway.
- Perform a count cut. Look at the bottom card (it is a number in the range 0-255). Count down from the top card that number+1. Cut after the card that you counted down to, leaving the bottom card on the bottom. The reason the last card is left in place is to make the step reversible. This is important for mathematical analysis of its security. A deck with a joker as the bottom card will remain unchanged by this step.
- Find the output card (and so the output number which is a byte). To do this, look at the top card. Count down that many cards+1. (Count

the top card as number one.) Write the card after the one you counted to on a piece of paper; don't remove it from the deck. (If you hit a joker, don't write anything down and start over again with step 1.) This is the first output card (and so the first byte). Note that this step does not modify the state of the deck.

* Keying the unkeyed deck: We will use as a key a string of arbitrary length, the characters being converted into numbers (bytes) by their ASCII codes. Start with the unkeyed deck. Perform the Solitaire operation, but instead of Step (6), do another count cut based on the character of the key. In other words, do step (5) a second time, using the character number as the cut number instead of the last card. Remember to put the top cards just above the bottom card in the deck, as before. Repeat the five steps of the Solitaire algorithm once for each character of the key. That is, the second time through the Solitaire steps use the second character of the key, the third time through use the third character, etc.

We denote by $Sol(K,n)$ the nth byte and by $Sol^k(K,n)$ the nth byte with at least k nonzero bits from the byte-sequence generated by the Solitaire algorithm keyed with the key K. Let $Bit_i(R)$ be ith bit of the byte R.

2. THE ALGORITHM

Consider the key K as a string of at least 8 characters (so this means that the key is at least 8 bytes). This will be called initial seed. Let $m = 2^8 = 256, r = 8, X_1 = Sol(K,1), ..., X_8 = Sol(K,8)$, so the initial seed generates the seed $X_1, ..., X_8$. Let also be $l \geq 2$ the "weight changing" period.

Then the pseudorandom sequence is given by the following recurrence for $n \geq 9$:

$$X_n = Bit_1(Sol^k(K, 9 + [\frac{n-9}{l}]))X_{n-8} + ... + Bit_8(Sol^k(K, 9 + [\frac{n-9}{l}]))X_{n-1} \mod 256$$

So we have a lagged Fibonacci generator of bytes given by a recurrence of order 8 modulo 256 (modulo a byte) with a periodically changing weight system generated by the Solitaire algorithm.

Our new pseudorandom generator will be called $LFibSol$, where $LFibSol^k(K,l,n)$ denotes the first n terms (bytes) of the pseudorandom sequence above.

3. THE ALGORITHM IN PSEUDOCODE

The pseudocode for our byte version of Solitaire is easily deducible from the first paragraph. Moreover the source code which only needs a slight modification can be found in [4]. So from now on we suppose that we already have the function $Sol^k(K,n)$ giving the nth byte with at least k nonzero bits from the byte-sequence generated by the Solitaire algorithm keyed with the key K.

THE ALGORITHM $LFibSol^m(K, l, n)$

INPUT: K - key; $l \geq 2$ - weight changing period; $k = 3, 4, 5$; $Sol^k(K, n)$
OUTPUT: n pseudorandom bytes
for $i := 1, 8$ **do** {generate 8 initial values}
  $X_i := Sol(K, i)$
end for
$R := Sol^k(K, 9)$ {the bits of R are the actual weight system controlling the recursion}
$p := 0$
$i := 10$
for $k := 1, n$ **do**
  if $p = l$ **then**{after l iterations, a new value is given to R}
    $R := Sol^k(K, i)$
    $i := i + 1$
  end if
  $X := 0$
  for $j := 1, 8$ **do**
    $X := X + X_j \cdot Bit(R, j) \mod 256$
  end for
  for $j := 2, 8$ **do**
    $X_{j-1} := X_j$
  end for
  $X_8 := X$
  $p := (p + 1) \mod (l + 1)$
  OUTPUT X
end for

4. SOME TEST RESULTS

To measure the quality of $LFibSol$ we used a statistical test suite for random and pseudo-random number generators for cryptographic applications, as described in NIST Special Publication 800-22 (with revisions dated May 15, 2001) (see [3]). The test suite performs the following 15 types of tests (as cited from [3]):

1. Frequency (Monobits) Test
Description: The focus of the test is the proportion of zeroes and ones for the entire sequence. The purpose of this test is to determine whether that number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to $\frac{1}{2}$, that is, the number of ones and zeroes in a sequence should be about the same.

2. Test for Frequency within a Block

Description: The focus of the test is the proportion of zeroes and ones within M-bit blocks. The purpose of this test is to determine whether the frequency of ones is an M-bit block is approximately $\frac{M}{2}$.

3. Cumulative Sum (Cusum) Test

Description: The focus of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted $(-1, +1)$ digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the random walk should be near zero. For non-random sequences, the excursions of this random walk away from zero will be too large.

4. Runs Test

Description: The focus of this test is the total number of zero and one runs in the entire sequence, where a run is an uninterrupted sequence of identical bits. A run of length k means that a run consists of exactly k identical bits and is bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence. In particular, this test determines whether the oscillation between such substrings is too fast or too slow.

5. Test for the Longest Run of Ones in a Block

Description: The focus of the test is the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Long runs of zeroes were not evaluated separately due to a concern about statistical independence among the tests.

6. Random Binary Matrix Rank Test

Description: The focus of the test is the rank of disjoint sub-matrices of the entire sequence. The purpose of this test is to check for linear dependence among fixed length substrings of the original sequence.

7. Discrete Fourier Transform (Spectral) Test

Description: The focus of this test is the peak heights in the discrete Fast Fourier Transform. The purpose of this test is to detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness.

8. Non-overlapping (Aperiodic) Template Matching Test

Description: The focus of this test is the number of occurrences of pre-defined target substrings. The purpose of this test is to reject sequences that exhibit too many occurrences of a given non-periodic (aperiodic) pattern. For this test and for the Overlapping Template Matching test, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. For this test, when the pattern is found, the window is reset to the bit after the found pattern, and the search resumes.

9. Overlapping (Periodic) Template Matching Test

Description: The focus of this test is the number of pre-defined target substrings. The purpose of this test is to reject sequences that show deviations from the expected number of runs of ones of a given length. Note that when there is a deviation from the expected number of ones of a given length, there is also a deviation in the runs of zeroes. Runs of zeroes were not evaluated separately due to a concern about statistical independence among the tests. For this test and for the Non-overlapping Template Matching test, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. For this test, when the pattern is found, the window again slides one bit, and the search is resumed.

10. Maurer's Universal Statistical Test

Description: The focus of this test is the number of bits between matching patterns. The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. An overly compressible sequence is considered to be non-random.

11. Approximate Entropy Test

Description: The focus of this test is the frequency of each and every overlapping m-bit pattern. The purpose of the test is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and $m+1$) against the expected result for a random sequence.

12. Random Excursions Test

Description: The focus of this test is the number of cycles having exactly K visits in a cumulative sum random walk. The cumulative sum random walk is found if partial sums of the $(0,1)$ sequence are adjusted to $(-1,+1)$. A random excursion of a random walk consists of a sequence of n steps of unit length taken at random that begin at and return to the origin. The purpose of this test is to determine if the number of visits to a state within a random walk exceeds what one would expect for a random sequence.

13. Random Excursions Variant Test

Description: The focus of this test is the number of times that a particular state occurs in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of occurrences of various states in the random walk.

14. Serial Test

Description: The focus of this test is the frequency of each and every overlapping m-bit pattern across the entire sequence. The purpose of this test is to determine whether the number of occurrences of the $2m$ m-bit overlapping patterns is approximately the same as would be expected for a random sequence. The pattern can overlap.

15. Linear Complexity Test

Description: The focus of this test is the length of a generating feedback register. The purpose of this test is to determine whether or not the sequence is complex enough to be considered random. Random sequences are characterized by a longer feedback register. A short feedback register implies non-randomness.

Since every random number generator failed test 11 (that is Approximate Entropy Test) every time, we will ignore Approximate Entropy failures. Due to lack of computational capacity we also had to ignore tests 8 and 9 (that is the Non-overlapping (Aperiodic) Template Matching Test and the Overlapping (Periodic) Template Matching Test).

We have empirically noticed that the algorithm $LFibSol^k(K, l, n)$ behaves particularly well in the case $k = 5$ and $l = 13$. We don't exclude the possibility of even better choices of constants. This is the subject of further research.

We explain now the testing procedure of $LFibSol^5(K, 13, 12500000)$ (here 12500000 is the length of the generated sequence, in bytes). Each of the tests checks 100 streams of n bits from $LFibSol^5(K, 13, 12500000)$ for any given key (initial seed) K. The values of n and the additional parameters are summarized in the following table (see [3] for further details).

Nr.	Test name	n(bits)	Add. param.
1.	Frequency (Monobit) Test	100000	
2.	Frequency Test within a Block	11000	M=128
3.	Cumulative Sums (Cusum) Test	100000	
4.	Runs Test	100000	
5.	Test for the Longest Run of Ones in a Block	1000000	
6.	Binary Matrix Rank Test	100000	
7.	Discrete Fourier Transform (Spectral) Test	100000	
10.	Maurer's "Universal Statistical" Test	1000000	
12.	Random Excursions Test	1000000	
13.	Random Excursions Variant Test	1000000	
14.	Serial Test	1000000	m=4
15.	Linear Complexity Test	1000000	M=500

The table below presents the test results for 20 randomly chosen keys. A pass is denoted by 1 and a failure by 0.

Seq. nr.	Key(Seed)	1	2	3	4	5	6	7	10	12	13	14	15
1.	[no key]	1	0	1	1	0	1	0	1	1	1	0	1
2.	dk2893hs	1	1	1	1	0	1	0	0	1	1	0	1
3.	j9dk6xa3	1	1	1	1	0	1	0	0	1	1	0	1
4.	aslw3vk9	1	0	1	1	0	1	0	0	1	1	0	1
5.	11sucwl8	1	1	1	1	0	1	0	0	0	1	0	1
6.	5pq7n2ye	1	0	1	1	0	1	0	1	1	1	0	1
7.	dj7sle8a	1	0	1	0	0	1	0	1	1	1	0	1
8.	cristina	1	1	1	1	0	1	0	0	1	1	0	1
9.	twmmb713	1	0	1	1	0	1	0	1	0	1	0	1
10.	dwwi6sle	1	1	1	1	0	1	0	0	1	1	0	1
11.	pTsTD7Qw	1	1	1	0	0	1	0	0	1	1	0	1
12.	1ieklsMW	1	1	1	1	0	1	0	0	1	1	0	1
13.	utKK1Zpk	1	1	1	0	0	1	0	0	1	1	0	1
14.	Vk5S9FHT	1	1	1	0	0	1	0	0	1	1	0	1
15.	Eoc28H4q	1	1	1	1	0	1	0	0	1	1	0	1
16.	GCEvmtfa	1	1	1	1	0	1	0	0	1	1	0	1
17.	mJ5zigDj	1	1	1	0	0	1	0	0	1	1	0	1
18.	kDGMqioe	1	1	1	1	0	1	0	0	1	1	0	1
19.	g7gGhvUo	1	1	1	1	0	1	0	1	1	1	0	1
20.	EpgSg4nu	1	1	1	1	0	1	0	0	1	1	0	1

The success rates for the 20 keys (initial seeds) are the following (here the percentage means the proportion of the keys passing the test, in other words the percentage of strong initial seeds for the given test):

Nr.	Test name	Success rate for 20 keys
1.	Frequency (Monobit) Test	100%
2.	Frequency Test within a Block	75%
3.	Cumulative Sums (Cusum) Test	100%
4.	Runs Test	75%
5.	Test for the Longest Run of Ones in a Block	0%
6.	Binary Matrix Rank Test	100%
7.	Discrete Fourier Transform (Spectral) Test	0%
10.	Maurer's "Universal Statistical" Test	25%
12.	Random Excursions Test	90%
13.	Random Excursions Variant Test	100%
14.	Serial Test	0%
15.	Linear Complexity Test	100%

We remark that failures in test 10 (Maurer's "Universal Statistical" Test) were very close to the limit between a pass and a failure. The results above give us hope that our PRNG is at least a medium level generator (see [5] for

a classification of different random number generators). Further research on the right choice of parameters may lead to an improvement.

Acknowledgements. The research for this paper was supported by grant PN2-P4-11-020/2007.

REFERENCES

[1] Janke, W., *Pseudo Random Numbers: Generation and Quality Checks*, Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes, J. Grotendorst, D. Marx, A. Muramatsu (Eds.), John von Neumann Institute for Computing, NIC Series, Vol. **10**, 447–458.
[2] http://en.wikipedia.org/wiki/Pseudorandom_number_generator
[3] http://csrc.nist.gov/groups/ST/toolkit/rng/stats_tests.html
[4] http://www.schneier.com/solitaire.html
[5] http://www.lavarnd.org/what/nist-test.html

Received December 1, 2008
Accepted March 11, 2009

"Babeş-Bolyai" University Cluj
Faculty of Mathematics and Computer Science
Str. M. Kogălniceanu nr.1.
RO-400084 Cluj-Napoca, România
E-mail: csacarea@math.ubbcluj.ro
E-mail: szanto.cs@gmail.com