“Cumbia” Pseudo-Random Number Generator Specifications

Consisting of the following:

4 LFSRs

1 of length 29

1 of length 27

1 of length 23

1 of length 31

My choice of tap polynomials is as follows:

x^29 + x^20 + x^16 + x^11 + x^8 + x^4 + x^3 + x^2 + 1

x^31 + x^27 + x^23 + x^19 + x^15 + x^11 + x^7 + x^3 + 1

x^23 + x^18 + x^16 + x^13 + x^11 + x^8 + x^5 + x^2 + 1

x^27 + x^22 + x^17 + x^15 + x^14 + x^13 + x^6 + x^1 + 1

The first concern I tried to take care of was of course brute force, for someone to try to brute force 2^110 is rather infeasible, unless there was a certain weakness to the way the generator works, so assuming there isn’t an alternative, brute force is quite slow while still allowing for a quite fast generator (generating 100 sequences of 1 million bits each took about 15 seconds, in java, and with code that could be optimized). And now for the actual “steps” to generate bits:

- Each register is assigned a position, either 0, 1, 2 or 3, at the beginning, register of size 29 is 0, the register of size 31 is 1 and so on.
- The output is the XOR of the bits that “fall off” from the front of registers 0 1 and 2, XOR 1, and the bit from register 3 simply gets thrown away.
- Once the output is generated, we now start “dancing” by finding each register a partner to “dance” with. We start from the top, and work our way down, first, register 0, no matter its size, its “partner bit” will be the 8^{th} bit, for register 1, it will be the 12^{th} bit, for register 2 it will be the 14^{th} bit, and for register 3 it will be the 18^{th} bit all of them starting from the left, we will look at the partner bit from register 0 and see if it matches with register 1, if it matches, we will swap these registers, meaning register 0 becomes now register 1 and the old register 1 is now register 0, and so on. A register may only swap or “dance” once per output bit, which is why we move down the order, then, if a register either has no partner it can swap to or all registers have swapped places, we are done and we go back to the beginning and output again using only the new first 3 registers and so on.
- After the fill is chosen, we should run the generator for a few thousand iterations (10000 in my case) and then we start generating the random numbers we should use.

Graphical Example after X number of iterations:

Register of length 29, starting at position 0:

0	1	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	1	1	1	0

Register of length 31, starting at position 1:

0	1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	0	0	1

Register of length 27, starting at position 2:

1	0	0	0	1	1	0	1	1	1	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	0

Register of length 23, starting at position 3:

0	1	1	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0	1	0	1	1	0

We now shift the registers, according to their polynomial.

In this case our output is 0 XOR 0 XOR 1 XOR 1 or simply 0 with 0,0,1 being the front bits of the first 3 registers XOR 1, and our partner bits are as follows:

Register 0: 0 (index 7, 8th bit)

Register 1: 1 (index 11, 12th bit)

Register 2: 0 (index 13, 14th bit)

Register 3: 0 (index 17, 18th bit)

We start from the top and check, and we find that registers 0 and 2 have a matching partner bit, so we swap them, register 1 doesn't have a matching partner, nor does register 3 so they stay where they are.

And now our generator looks like this:

1

Register of length 27, now at position 0:

0	0	0	1	1	0	1	1	1	0	1	0	0	1	1	0	0	0	0	0	1	0	1	1	0	0	0

Register of length 31, still at position 1:

1	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1	1	1	1	0	0	0	0	1	1	1	0	0	1	1

Register of length 29, now at position 2:

1	0	0	1	1	1	0	0	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	1	1	1	0	1

Register of length 23, still at position 3:

1	1	0	1	1	1	1	1	0	1	0	0	1	0	0	0	0	1	0	1	1	0	1

And we repeat this process every time we want to generate a new bit.

This LFSR passes all the NIST tests, however, it seems to perform worst on “NonOverlappingTemplate” test, having occasionally about .96 proportion on some of these. There is 100ish rows for this test in each Analysis report, and for most of the test it shows either 99 or 100, like pictured below:

```
 10   8  11  11  17   9   9  10   8   7  0.637119     98/100     NonOverlappingTemplate
 11   9  10  12   6  11  10  13  12   6  0.816537     99/100     NonOverlappingTemplate
 11  12  11   8  11   8   9   8   7  15  0.798139     98/100     NonOverlappingTemplate
 11  12   8  10  11   9   9   8   7  15  0.834308     98/100     NonOverlappingTemplate
  6   8   9  12   9   7   7  10  15  17  0.224821     99/100     NonOverlappingTemplate
 11   9   9  18  10  12  10   7   7   7  0.366918     99/100     NonOverlappingTemplate
  7  10  12  10   8  11   9  11  16   6  0.616305     99/100     NonOverlappingTemplate
 13  13  12  11   6  15   9   6   7   8  0.401199     99/100     NonOverlappingTemplate
  9  13  15  10   3  17   8  10   6   9  0.080519     99/100     NonOverlappingTemplate
  8  14  12   9  13   7   4  10  12  11  0.494392     96/100     NonOverlappingTemplate
 12  11   6  14   7   6   6   7  15  16  0.096578     98/100     NonOverlappingTemplate
  7  17   7   9   6  10  13  15  10   6  0.145326     99/100     NonOverlappingTemplate
 16   5  13   8   8   9   9  12   9  11  0.474986    100/100     NonOverlappingTemplate
```

Since the passing rate is 96/100 for 100 bitstreams tested, I consider this a success, as for the other tests, for all the files I have generated, having 100 strings of 1 million bits each, it has never been below .97 or 97/100 proportion for any of the other tests, so I would consider the generator to pass all the other tests as well.

It is also important to note that the generator will work with the LFSRs placed in any order, so if we wanted to increase security, part of the cryptographic key could be the order of the LFSRs as well.

This is how my analysis report looks like for a sample 100 streams of 1 million bits each:

RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES

---

generator is <data/randomNumbers.txt>

C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 P-VALUE PROPORTION STATISTICAL TEST

---

14 19 8 6 9 9 9 10 6 10 0.137282 97/100 Frequency

8 9 9 14 9 14 10 9 8 10 0.883171 99/100 BlockFrequency

16 12 8 14 11 6 7 11 5 10 0.262249 97/100 CumulativeSums

13 18 13 9 8 7 7 9 10 6 0.202268 98/100 CumulativeSums

7 11 12 14 11 7 8 10 6 14 0.574903 100/100 Runs

8 12 7 6 8 15 7 14 12 11 0.419021 99/100 LongestRun

9 9 9 8 12 7 12 11 11 12 0.964295 98/100 Rank

12 11 13 14 6 8 6 11 9 10 0.657933 98/100 FFT

9 13 10 7 12 10 12 9 9 9 0.964295 100/100 NonOverlappingTemplate

15 8 8 9 4 9 12 13 12 10 0.455937 98/100 NonOverlappingTemplate

8 13 12 16 9 9 13 4 6 10 0.236810 99/100 NonOverlappingTemplate

11 8 6 9 16 7 7 11 12 13 0.437274 99/100 NonOverlappingTemplate

9 11 13 10 9 8 7 12 9 12 0.946308 100/100 NonOverlappingTemplate

…100 more NonOverlappingTemplate rows which are omitted.

5 14 7 6 11 17 9 4 17 10 0.016717 99/100 OverlappingTemplate

11 7 9 8 8 7 10 17 13 10 0.474986 100/100 Universal

7 9 8 14 7 10 7 10 10 18 0.262249 99/100 ApproximateEntropy

…all RandomExcursionsVariant and RandomExcursions rows are 56/56 so I also omitted.

4 6 3 7 7 11 6 3 7 2 0.137282 56/56 RandomExcursions

6 6 6 0 9 9 6 4 6 4 0.171867 56/56 RandomExcursionsVariant

12 11 10 9 9 10 8 10 10 11 0.998821 98/100 Serial

7 9 8 15 10 16 9 9 9 8 0.514124 100/100 Serial

15 6 10 6 7 10 11 15 9 11 0.401199 99/100 LinearComplexity

- -

The minimum pass rate for each statistical test with the exception of the

random excursion (variant) test is approximately = 96 for a

sample size = 100 binary sequences.

The minimum pass rate for the random excursion (variant) test

is approximately = 53 for a sample size = 56 binary sequences.

According to this, on average, most of the bitstreams would pass all of the NIST statistical tests.