# Lamar: A New Pseudorandom Number Generator Evolved by Means of Genetic Programming

Carlos Lamenca-Martinez, Julio Cesar Hernandez-Castro, Juan M. Estevez-Tapiador, and Arturo Ribagorda

Computer Science Department, Carlos III University of Madrid
{clamenca, jcesar, jestevez, arturo}@inf.uc3m.es

**Abstract.** Pseudorandom number generation is a key component of many Computer Science algorithms, including mathematical modeling, stochastic processes, Monte Carlo simulations, and most cryptographic primitives and protocols. To date, multiple approaches that use Evolutionary Computation (EC) techniques have been proposed for designing useful Pseudorandom Number Generators (PRNGs) for certain non-cryptographic applications. However, none of the proposals have been secure nor efficient enough to be of interest for the much more demanding crypto world. In this work, we present a general scheme, which uses Genetic Programming (GP), for the automatic design of crypto-quality PRNGs by evolving highly nonlinear and extremely efficient functions. A new PRNG named Lamar and obtained using this scheme is proposed, whose C code and preliminary security analysis are provided.

## 1 Introduction

There are two types of random number generators: True Random Number Generators (TRNGs), also known as Random Number Generators (RNGs), that get their randomness from natural sources; and Pseudorandom Number Generators (PRNGs), based on deterministic algorithms that expand short keys into sequences of seemingly random bits.

Hardware implementations of TRNGs, like the chaotic system proposed in [32], are generally costly and slow, so their use is limited to only a few cases like the generation of seeds for PRNGs in cryptographic applications. That is why there is a constant need for powerful and efficient PRNGs, to be implemented in hardware and software.

Designing new, better, and more efficient PRNGs is an important open problem in Computer Science and, in particular, in computer security and cryptography. Apart from that, they play a major role in areas such as mathematical modeling, stochastic processes, and Monte Carlo simulation.

Unfortunately, it is not possible to prove randomness, because there is no efficient and deterministic definition of this rather abstract idea. Some basic

T.P. Runarsson et al. (Eds.): PPSN IX, LNCS 4193, pp. 850–859, 2006.

concepts that help in detecting evidence against the randomness hypothesis can be consulted in [11]. In 2001, the National Institute of Standards and Technology (NIST) proposed a comprehensive suite of randomness tests for the evaluation of PRNGs used in cryptographic applications [25]. One of the most stringent sets of randomness tests is Diehard [14], developed in 1996 by Marsaglia who, in 2002, extended and improved it by including the new tests presented in [15]. On the other hand, ENT [30] is a relatively light but very quick battery of tests. Nevertheless, none of these tests suites ensure, when successfully passed, that a given generator is useful for all kinds of applications. Systematically passing the NIST and Diehard batteries of tests provides, however, evidence in favor of a good degree of output randomness, probably adequate to make the algorithm suitable even for the most demanding application a PRNG can have: cryptography.

Multiple techniques based on EC (being Cellular Automata, CA, the most successful approach due to its output complexity and parallel nature) have been proposed for designing useful PRNGs for certain non-cryptographic applications. Nevertheless, the results typically do not pass a battery of very demanding statistical tests or, in the few cases when they do, it is at a high computational cost, resulting in very slow and completely worthless schemes for real-world high-throughput applications.

In this work, we present a general GP scheme for the automatic design of crypto-quality PRNGs by evolving highly nonlinear and extremely efficient functions. As an example, we propose Lamar, a new PRNG obtained using this approach, whose C code and preliminary security analysis are provided.

## 2 State of the Art

Many EC paradigms have been used for designing non-cryptographic PRNGs: Genetic Algorithms, GAs [6]; GP [5,12]; CA [31]; Cellular Programming [28], etc. However, none of the proposals were secure nor efficient enough to be of interest for the much more demanding crypto applications. With a different approach, some works [3,18] study the behavior of EC algorithms to measure the quality of PRNGs and, although this is a quite interesting research line, it is still wide open and waiting for mature results.

Randomness is not the only requirement for crypto-quality PRNGs, but just one more in a long list of very demanding properties. That is why very few works have used EC for developing cryptographic primitives. In [22], GAs are applied to generate boolean functions with excellent cryptographic properties, particularly a high degree of nonlinearity. A key-exchange protocol developed by means of neural networks was presented in [10]. We should also mention [27], where a new block cipher relying on a reversible CA was proposed. CAs have also been used in the design of stream ciphers [21] and hash functions [8,20]. Nevertheless, many of the previous schemes have been broken [2]. Despite these discouraging results, a steady flow of new EC-based cryptographic primitives (especially PRNGs and symmetric ciphers) exists, although they are looked upon suspiciously by the vast majority of the crypto community.

## 3 Avalanche Effect

Nonlinearity, as randomness, has not a complete, unique, and satisfactory definition. Fortunately, we will just try to measure a very specific mathematical property named the Avalanche Effect. This property reflects, to some extent, the intuitive idea of high nonlinearity: a very small difference in the input produces an avalanche of changes in the output, hence its name. Formally, let $H(x, y)$ be the Hamming distance between $x$ and $y$. It is said that $F : \{0,1\}^m \rightarrow \{0,1\}^n$ has the Avalanche Effect property when:

$$\forall x, y | H(x, y) = 1, \quad Average\left(H(F(x), F(y))\right) = \frac{n}{2}$$

That is, a minimum random input change (one single bit) produces a maximum output change (half of the bits), on average. This definition also tries to reflect the general concept of independence between input and output (a good reason for being the base for our proposal, and an intuitive explanation for its applicability to the generation of PRNGs). An ideal $F$ will resemble a perfect random function and will have a perfect Avalanche Effect. So, it seems natural to look for such functions by optimizing the amount of Avalanche Effect.

We will use, in fact, an even more demanding property: the Strict Avalanche Criterion [4], which can be mathematically described as:

$$\forall x, y | H(x, y) = 1, \quad H(F(x), F(y)) \sim B\left(n, \frac{1}{2}\right)$$

That is, the Hamming distances, after changing one single input bit, follow a Binomial distribution with parameters $n$ and $1/2$. It is interesting to note that this property implies the Avalanche Effect, because the mean of such a random variable is $n/2$.

## 4 Experimentation Issues

We have used the `lil-gp` library [1] for our experimentation. Next, we describe the parameters the user has to adjust in order to define a particular problem.

### 4.1 Function Set

As functions are the building blocks of the individuals we will obtain, the correct definition of this set is critical to our problem. We decided to include only very efficient operations, easy to implement both in hardware and software and common in other implementations of PRNGs and stream ciphers. Hence, the inclusion of basic operations such as **vrotd** (one-bit right rotation), **vroti** (one-bit left rotation), **xor** (addition mod 2), **and** (bitwise and), **or** (bitwise or), and **not** (bitwise not) is an obvious choice. The **sum** (sum mod $2^{32}$) operator is also useful, in order to avoid linearity.

We did not include **mult** (multiplication mod $2^{32}$) at first because, depending on the implementation, the multiplication of two 32-bit values can be up to fifty times costlier than an **and** operation [7]. After extensive experimentation, we concluded that its inclusion was beneficial. Furthermore, there are some widely-used cryptographic primitives that use multiplication, like RC6 [24].

In some architectures, **vrotd** and **vroti** are also quite inefficient. We tried to solve this by including $\gg$ (one-bit right shift) and $\ll$ (one-bit left shift) too. These four operators were also implemented in a binary form (**vrotdk**, **vrotik**, **rotdk**, and **rotik**, respectively), where the argument $k$ represents the number of positions that the first argument must be *moved*.

Although some operators were never used in any of the best individuals found (for example, the **not** and the **and** operators never showed up), when we included *all of them* in the function set, the results were notably better. That is why we decided to let the evolution discard the useless ones.

### 4.2 Terminal Set

The terminals will be eight 32-bit unsigned integers ($a_0$, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$, $a_6$, $a_7$) for representing a 256-bit input. We also included Ephemeral Random Constants (ERCs), which are constant values (in our problem, 32-bit random values) that the GP system uses to generate better individuals. The idea behind this is providing a constant value, independently of the input.

### 4.3 Fitness Function

We wanted the PRNG to be efficient, complex, and robust because our objective is including it in the scheme of a stream cipher. To achieve this, we used:

$$Fitness = 10^9/\chi^2 \tag{1}$$

where $\chi^2$ is the $\chi^2$ goodness-of-fit test statistic that measures the proximity of the computed Hamming distances distribution to the sought theoretical $B(n, 1/2)$. It was necessary to amplify the fitness function (multiplying by $10^9$) because the initial values of the $\chi^2$ statistic were extremely high, making the fitness negligible at the beginning of the evolution process.

Summarizing, the fitness of each individual is computed as follows: we use the Mersenne Twister generator [17] to randomly generate the tuple ($a_0$, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$, $a_6$, $a_7$). The output $O_0$ corresponding to this input is stored. Then, we randomly flip one single bit of this 256-bit input and obtain its output $O_1$. Now, we store the Hamming distance between these two output values, $H(O_0, O_1)$. This process is repeated a number of times ($2^{14} = 16384$ was experimentally proved to be enough) and, each time, a Hamming distance between 0 and 32 is obtained. Therefore, the fitness of each individual is computed by using the $\chi^2$ goodness-of-fit test statistic that measures the distance between the observed distribution of the Hamming distances and their theoretical distribution under perfect Strict Avalanche Criterion hypothesis ($B(32, 1/2)$). Thus, our GP system tries to minimize this statistic in order to maximize *Fitness* (1).

### 4.4 Tree Size Limitations

When using GP, the depth and/or the number of nodes of the individuals should be limited. We tried both limiting the depth and not limiting the number of nodes, and vice versa. The best results were obtained by using the latter option, that is, we allowed the PRNG to use up to 100 nodes for trying to ensure a high degree of Avalanche Effect and robustness, while keeping a relatively small size, compatible with efficiency.

### 4.5 Results

We ran 20 experiments with different seeds for generating the initial population ($seed_i = (\pi * 100000)^i \pmod{1000000}$), a population size of 150 individuals, a crossover probability of 0.8, a reproduction probability of 0.2, and an ending condition of reaching 250 generations. These parameters were experimentally found to be adequate for our purposes.

Next, we show the tree corresponding to the best individual found over these experiments. This PRNG has an Avalanche Effect of 15.9631 (randomly flipping one input bit, the 32-bit output changes in 15.9631 bits on average, being 16.00 the optimal value) and presents a $\chi^2$ goodness-of-fit test statistic of 12.6614 for a $\chi^2$ probability distribution with 32 degrees of freedom; it means that, with probability 0.999112, the computed Hamming distances come from a Binomial distribution $B(32, 1/2)$.

```
=== BEST-OF-RUN ===
              generation: 153
                   nodes: 100
                   depth: 27
                    hits: 159631
TOP INDIVIDUAL: -- #1 --
                    hits: 159631
             raw fitness: 6237837.6345
    standardized fitness: 6237837.6345
        adjusted fitness: 6237837.6345
TREE:
 (sum (sum a5 (xor a6 a1)) (sum (vrotd 928a463)
 (mult (sum a5 (mult (sum a7 (sum a5 (mult
 (xor a3 (xor (vrotdk (xor (xor (rotdk (xor (xor
 (rotdk (sum a5 (mult (xor (rotdk (sum a3 (mult
 (xor (xor a6 a2) (xor (vroti (xor a6 a1)) a4))
 928a463)) (vrotd 928a463)) (roti a1)) 928a463))
 928a463) a1) (mult a3 a6)) 928a463) a1) (mult a3
 (roti a1))) 928a463) (sum (rotd (vroti (xor (xor
 (rotdk (sum a5 (mult (xor (rotdk (sum a5 (mult
 (xor a0 (xor (xor a3 a1) a7)) 928a463)) (vrotd
 928a463)) a0) 928a463)) (vrotd 928a463)) a0) a2)))
 (xor a6 a2)))) 928a463))) 928a463)) 928a463)))
```

## 5 Security Analysis

In this section, the results of testing the statistical and cryptographic properties of Lamar are presented.

### 5.1 Statistical Properties

We have performed a preliminary security analysis of our PRNG, consisting in examining the statistical properties of its output over a low-entropy input. In our case, we have set the 256-bit input to a simple incremental counter starting at zero ($a_i^n = i + 8 * (n - 1)$).

Following this scheme, we have generated a file of 250MB to be analyzed with three batteries of statistical tests: ENT, Diehard, and NIST. The results obtained with ENT and Diehard are presented in Tables 1 and 2, respectively. In the latter, when several p-values were produced in the same test, we summarized them by a Kolmogorov-Smirnov p-value (marked with *), being necessary it to be greater than 0.05 to be considered successful. Lamar also passed the very demanding – because it is oriented to cryptographic applications– NIST statistical battery. We have computed 100 p-values for every test in the statistical suite; the proportion of successful ones is presented in Table 3. If this proportion is lower than 0.96, it is considered that the whole test failed. From these results, we can conclude that the output successfully passed all the randomness tests, even with the output being obtained from an extremely low-entropy input.

**Table 1.** Results obtained with ENT

| Test | Result |
|---|---|
| Entropy | 7.999999 bits/byte |
| Compression Rate | 0% |
| $\chi^2$ Statistic | 258.29 (50%) |
| Arithmetic Mean | 127.4984 |
| Monte Carlo $\pi$ Estimation | 3.141524737 (0.00%) |
| Serial Correlation Coefficient | -0.000060 |

### 5.2 PRNG Quality Evaluation

To compare how Lamar performs against several different PRNGs, we used Johnson's scoring scheme [9]: we initialized ($a_0$, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$, $a_6$, $a_7$) with 32 different random 256-bit values obtained from `http://randomnumber.org`, got 32 different 10MB files, and then assigned scores based on the results of the Diehard tests. The terminal updating during this process was made in a feedback manner, that is:

$$a_i^{n+1} = a_{i+1}^n \quad \forall i = 0, ..., 6$$
$$a_7^{n+1} = O^n_{(a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7)}$$

The PRNGs we have compared to ours are of several different kinds: Linear Congruential Generators (rand [11], rand1k [19], pm [23]), Multiply with Carry Generators (mother [13]), Additive and Substractive Generators (add [11], sub [23]), Compound Generators (shsub [11], shpm [23], shlec [23]), Feedback Shift Register Generators (tgfsr [16], fsr [26]), and Tausworthe Generators (tauss [29]).

**Table 2.** Test results obtained with the Diehard suite

| Test | p-value | Test | p-value |
|---|---|---|---|
| Birthday Spacings | 0.909* | Count the 1s in a Stream of Bytes | 0.894001 |
| GCD | 0.816* | Count the 1s in Specific Bytes | 0.857* |
| Overlapping Permutations | 0.976* | Parking Lot Test | 0.476150 |
| Ranks of 31×31 and 32×32 Matrices | 0.964* | Minimum Distance Test | 0.789136 |
| Ranks of 6×8 Matrices | 0.354932 | Random Spheres Test | 0.676330 |
| Monkey Tests on 20-bit Words | 0.759* | The Squeeze Test | 0.756284 |
| Monkey Test OPSO | 0.236* | Overlapping Sums Test | 0.245735 |
| Monkey Test OQSO | 0.613* | Runs Up and Down Test | 0.379 |
| Monkey Test DNA | 0.793* | The Craps Test | 0.985* |
| **Overall p-value** | | **0.360260** | |

**Table 3.** Proportion results obtained with the NIST suite

| Test | Proportion | Test | Proportion |
|---|---|---|---|
| Frequency | 1.0000 | Random-excursions | 1.0000, 0.9718 |
| Block-frequency | 1.0000 | | 0.9859, 0.9859 |
| Cumulative-sums | 1.0000, 1.0000 | | 1.0000, 1.0000 |
| Runs | 1.0000 | | 1.0000, 1.0000 |
| Longest-run | 0.9900 | Random-excursions-variant | 1.0000, 1.0000, 1.0000 |
| Rank | 0.9900 | | 1.0000, 1.0000, 0.9718 |
| FFT | 0.9700 | | 0.9859, 0.9859, 1.0000 |
| Overlapping-templates | 1.0000 | | 1.0000, 1.0000, 1.0000 |
| Universal | 0.9900 | | 0.9859, 0.9718, 0.9859 |
| Apen | 0.9900 | | 0.9859, 0.9859, 0.9718 |
| Linear-complexity | 1.0000 | Serial | 0.9900, 0.9900 |

Each of the Diehard tests produces one or more p-values. We categorize them as good, suspect or rejected. We classify a p-value as rejected if $p \geq 0.998$, and as suspect if $0.95 \leq p < 0.998$; all other p-values are considered to be good. We assign two points for every rejection, one point for every suspect classification, and no points for the rest. Finally, we add up these points to produce a global Diehard score for each PRNG, and compute the average over the 32 evaluations: low scores indicate good PRNG quality. The information relating to the different PRNGs was taken from [18,19].

The results are presented in Table 4, where the behavior of a random variable $X$ following the distribution of the scores ($X = 0$ with probability 0.95, $X = 1$ with probability 0.048, and $X = 2$ with probability 0.002) is also included. We note that Lamar is outstandingly better than the rest of the analyzed PRNGs: the lowest scores correspond to shsub (17.125) and fsr (17.90625), significantly greater than Lamar's (11.78125). On the other hand, the average scores increase up to 50.59375 (pm), 66.53125 (rand), and even 291.78125 (rand1k). As these are non-cryptographic PRNGs, this behavior could have been foreseen.

To measure the proximity between Lamar and the true random variable $X$, a $\chi^2$ test has been computed, which is also shown in Table 4. As a result, we can affirm the behavior of Lamar can not be distinguised from that of a random variable at a significance level of $\alpha = 0.99$.

**Table 4.** PRNG Diehard Scores

| PRNG | Total Score | Mean | |
|---|---|---|---|
| rand | 2129 | 66.53125 | |
| rand1k | 9337 | 291.78125 | |
| pm | 1619 | 50.59375 | |
| mother | 602 | 18.8125 | |
| add | 577 | 18.03125 | |
| sub | 655 | 20.46875 | |
| shsub | 548 | 17.125 | |
| shpm | 799 | 24.96875 | |
| shlec | 751 | 23.46875 | |
| fsr | 573 | 17.90625 | |
| tgfsr | 584 | 18.25 | |
| tauss | 935 | 29.21875 | |
| **Lamar** | **377** | **11.78125** | $\chi^2$ **Test p-value** |
| **Random Variable** | **371.072** | **11.596** | **0.021** |

## 6 Conclusions and Future Work

In this work, we have proved that the GP paradigm can be successfully applied to the design of competitive PRNGs. The most relevant aspect of our scheme is the selection of the fitness function, where nonlinearity and efficiency are the paramount objectives. We have opted to use the Strict Avalanche Criterion as the key property of the explored mathematical functions, and used only very efficient operators to construct them.

The proposed PRNG has successfully passed several batteries of very demanding statistical tests, which were not part of the fitness function; being this, by itself, quite an interesting result. Although passing these tests does not ensure a certain security level, it guarantees, to some extent, that neither trivial weaknesses nor implementation bugs exist. Moreover, Lamar has been compared to several PRNGs by measuring their statistical quality from Diehard results, and we have proved ours is the best and that there is no evidence to ensure Lamar is different from a random variable.

Future work will use Lamar as the building block of a stream cipher. For this, a deeper security analysis is needed, particularly against basic cryptanalytic attacks. We have tried to incorporate robustness in Lamar by construction, even though further testing is required. Additionally, speed tests should be performed to measure the exact efficiency of the proposed generator. In particular, it could be interesting to compare the efficiency of Lamar, which is expected to be good by design, with the current state-of-the-art stream ciphers, including RC4 and all its variants.

## References

1. The `lil-gp` GP system. http://garage.cps.msu.edu/software/lil-gp/.
2. F. Bao. Cryptanalysis of a partially known cellular automata cryptosystem. *IEEE Trans. on Computers*, 53(11):1493–1497, 2004.

3. E. Cantú-Paz. On random numbers and the performance of genetic algorithms. In *Proc. of GECCO'02*, volume 2, pages 311–318. Morgan Kaufmann, 2002.
4. R. Forré. The strict avalanche criterion: Spectral properties of boolean functions and an extended definition. In *Proc. of CRYPTO'88*, LNCS, pages 450–468. Springer-Verlag, 1990.
5. J.C. Hernandez-Castro, P. Isasi, and A. Seznec. On the design of state-of-the-art PRNGs by means of genetic programming. In *Proc. of the IEEE CEC'04*, pages 1510–1516. IEEE Press, 2004.
6. J.C. Hernandez-Castro, A. Ribagorda, P. Isasi, and J.M. Sierra. Finding near optimal parameters for linear congruential PRNGs by means of evolutionary computation. In *Proc. of GECCO'01*, pages 1292–1298. Morgan Kaufmann, 2001.
7. G. Hinton et al. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.
8. S. Hirose and S. Yoshida. A one-way hash function based on a two-dimensional cellular automaton. In *Proc. of the 20th Symposium on Information Theory and its Applications*, volume 1, pages 213–216. Matsuyama, 1997.
9. B.C. Johnson. Radix-b extensions to some common empirical tests for PRNGs. *ACM Trans. on Modeling and Comp. Sim.*, 6(4):261–273, 1996.
10. I. Kanter, W. Kinzel, and E. Kanter. Secure exchange of information by synchronization of neural networks. *Europhysical Letters*, 57(141), 2002.
11. D.E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.
12. J.R. Koza. Evolving a computer program to generate random number using the genetic programming paradigm. In *Proc. of the 4th Int. Conference on Genetic Algorithms*, pages 37–44. Morgan Kaufmann, 1991.
13. G. Marsaglia. Yet another RNG. Posted to sci.stat.math, 1994.
14. G. Marsaglia. *The Marsaglia Random Number CDROM Including the DIEHARD Battery of Tests of Randomness*. http://stat.fsu.edu/pub/diehard, 1996.
15. G. Marsaglia and W.W. Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3), 2002.
16. M. Matsumoto and Y. Kurita. Twisted GFSR generators. *ACM Trans. on Modeling and Comp. Sim.*, 2(3):179–194, 1992.
17. M. Matsumoto et al. Mersenne twister: A 623-dimensionally equidistributed uniform PRNG. *ACM Trans. on Modeling and Comp. Sim.*, 8(1):3–30, 1998.
18. M.M. Meysenburg and J.A. Foster. The quality of PRNGs and simple genetic algorithm performance. In *Proc. of the 7th Int. Conference on Genetic Algorithms*, pages 276–281. Morgan Kaufmann, 1997.
19. M.M. Meysenburg and J.A. Foster. Randomness and GA performance, revisited. In *Proc. of GECCO'99*, volume 1, pages 425–432. Morgan Kaufmann, 1999.
20. M. Mihaljevic, Y. Zheng, and H. Imai. A cellular automaton based fast one-way hash function suitable for hardware implementation. volume 1431 of *LNCS*, pages 217–233. Springer-Verlag, 1998.
21. M.J. Mihaljevic. An improved key stream generator based on the programmable cellular automata. In *Proc. of ICICS'97*, volume 1334 of *LNCS*, pages 181–191. Springer-Verlag, 1997.
22. W. Millan, A. Clark, and E. Dawson. An effective genetic algorithm for finding boolean functions. In *Proc. of ICICS'97*, 1997.
23. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.
24. R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 block cipher, v1.1, August 20 1998.

25. A. Rukhin et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22, http://csrc.nist.gov/rng/, 2001.
26. B. Schneier. *Applied Cryptography.* John Wiley and Sons, 1994.
27. M. Seredynski and P. Bouvry. Block cipher based on reversible cellular automata. *Next Generation Computing Journal*, 23(3):245–258, 2005.
28. M. Sipper and M Tomassini. Generating parallel random number generators by cellular programming. *Int. Journal of Modern Physics C*, pages 181–190, 1996.
29. S. Tezuka and P. L'Ecuyer. Efficient and portable combined Tausworthe Random Number Generators. *ACM Trans. on Modeling and Comp. Sim.*, 1(2):99–112, 1991.
30. J. Walker. *ENT Randomness Tests.* http://www.fourmilab.ch/random/, 1998.
31. S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169, 1986.
32. M.E. Yalcin, J.A.K. Suykens, and J. Vandewalle. True random bit generation from a double-scroll attractor. *IEEE Trans. on Circuits and Systems-I: Regular Papers*, 51(7):1395–1404, 2004.

## Appendix: C Source Code

Finally, a C implementation of the Lamar PRNG is included, where `unsigned long` variables represent unsigned integers of 32 bits.

```
unsigned long lamar (unsigned long a0, unsigned long a1, unsigned long a2, unsigned long a3,
                     unsigned long a4, unsigned long a5, unsigned long a6, unsigned long a7)
{  unsigned long nonce, aux1, aux2, aux3;

   aux1 = ((vroti(a6 ^ a1) ^ a2 ^ a4 ^ a6) * 0x928a463) + a3;
   aux2 = rotdk(aux1, vrotd(0x928a463)) ^ (a1<<1);
   aux1 = (0x928a463 * aux2) + a5;
   aux2 = rotdk(aux1, 0x928a463) ^ a1 ^ (a3 * a6);
   aux1 = rotdk(aux2, 0x928a463) ^ a1 ^ ((a1<<1) * a3);
   aux2 = ((a0 ^ a1 ^ a3 ^ a7) * 0x928a463) + a5;
   aux3 = ((rotdk(aux2, vrotd(0x928a463)) ^ a0) * 0x928a463) + a5;
   aux2 = rotdk(aux3, vrotd(0x928a463)) ^ a0 ^ a2;
   aux3 = (a2 ^ a6) + (vroti(aux2)>>1);
   aux2 = vrotdk(aux1,0x928a463) ^ a3 ^ aux3;
   aux1 = ((aux2 * 0x928a463) + a5 + a7) * 0x928a463;
   aux2 = (a6 ^ a1) + a5 + vrotd(0x928a463);
   nonce = ((aux1 + a5) * 0x928a463) + aux2;
   return(nonce);  }
```