Introduction to Secure PRNGs

Majid Babaei, Mohsen Farhadi
Department of Computer Engineering, Shahrood University of Technology, Shahrood, Iran
E-mail: babaee@comp.tus.ac.ir, mfarhadi@shahroodut.ac.ir

Abstract

Pseudo-Random Number Generators (PRNGs) are required for generating secret keys in cryptographic algorithms, generating sequences of packet in Network simulations (workload generators) and other applications in various fields. In this paper we will discuss a list of some requirements for generating a reliable random sequence and then will present some PRNG methods which are based on combinational chaotic logistic map. In the final section after a brief introduction to two statistical test packets, TestU01 and NIST suite tests, the PRNG methods which are presented in the fourth section will be appraised under these test packets and the results will be reported.

Keywords: Cryptography, Chaotic Random Bit Generator, NIST Test Suite, TestU01

1. Introduction

Pseudo-random number generators (PRNGs) are useful in many applications such as the crypto-system in network communication [1] also one of the important methods in simulation is Mont Carlo [2] which needs to generate very high quality pseudo-random sequence and calculation of numerical integration which is not solved by regular methods [3-5].

Regarding the threats to the security [3-16], the statistical quality of PRNGs is becoming more important than before. For example a super computer might generate $10^9$ random numbers per second and the cryptography algorithm needs $10^{10}$ random numbers to create a secure channel in a very important communications, thus small correlation or other weaknesses in the generated sequence could easily lead to the critical leak in several network layers.

The distributions should be prepared based on their commercial applications such as “normal”, “exponential”, “poisson” and so on. But in this paper we consider only the generation of uniformly distributed numbers. In more details, we will focus on the real number sequence which is uniformly distributed on the interval $(0...1)$.

The basic point in generating pseudo-random number is that these generators are deterministic because the digital computers are not able to generate truly random numbers. So the statistical test needs to be presented and the PRNGs should pass a number of important statistical tests, before being released for security usage in communication networks.

In this paper, in Section 2, overview of using of chaotic maps as the reliable PRNG will be presented. Then in Section 3 some requirements for generating a high quality PRNG will be discussed. In Section 4 some class of improving PRNGs will be presented and finally in Section 5 two strong statistical test packages (i.e. NIST suite tests and TestU01) will be introduced and all PRNG methods which are tested by author, will be purposed and the results will be reported into various tables.

2. Overview

Generation random numbers by combinational chaotic maps is one of the best methods to improve statistical properties. For example, in 2006, Wang et al. proposed a pseudorandom number generator based on a z-logistic map [4]. In 2007, Ergun and Ozogur proposed the novel random bit sequence of a non-autonomous chaotic electronic circuit [5]. Then, Hu et al. proposed a true random number generator by computer mouse movement [6]. In 2009 Patidar et al. designed a random bit generator based on two chaotic logistic maps which is generated by comparing the outputs of the both chaotic logistic maps [7]. Recently in B. Fechner and A. Osterloh presented a meta-level true random number generator [8].

3. Generating High Quality PRNG

In many papers there have been discussions about the
requirements for a good PRNG [3-16]. In this section these attempts will be summarized and briefly described.

3.1. Uniformity of Distribution

Uniformity of distribution is the main concern in the statistical tests of random sequences. It means that at all points in the generation of a sequence of random or pseudo-random bits, the probability of Zero or One is as much as the probability of nine [9].

3.2. Independence

Each part of a random sequence should be independent from other parts. In the sample sequence \((m_0, m_1, \ldots)\), random numbers generate in the \(d\)-dimensional space so the sample part of this sequence:

\[
d-tuples = m_{dm}, m_{dm+1}, \ldots, m_{dm+d-1}
\]

Which is uniformity distribution should be independent in the \(d\)-dimensional cube \([0,1]^d\). For example in Halton sequence [10] which is the low discrepancy method [10] to generate random numbers. Figure 1 decreased independence by increasing dimension and become totally dependent in Figure 2. Also this multidimensional clustering is clear in high dimensional in the Sobol sequence [11] (Figures 3-4).

3.3. Efficient Length of Period

Some classic algorithms for PRNGs such as Middle Square method and middle product method, although have the unique characteristics to generate pseudo-random numbers, but not enough length of periods.

In 2010 this problem solved by H. Rahimov, M. Babaei and H. Hassanabadi [12].

3.4. Unpredictability

Unpredictability is one of the important points in cryptography, because the random sequence with these advantages (i.e. efficient length of period, good independency and uniformity of distribution) could be predictable thus existing a lot of threats in the Secret communication, we need to generate the sequence unpredictable.

In the sample sequence \((m_0, m_1, \ldots, m_n)\), in the best conditions if a group of hackers have the largest part of this sequence (i.e. \(m_0, m_1, \ldots, m_{n-1}\)) they may guess the other parts. But in the unpredictable PRNG They are not able to guess \(m_n\) with probability more than 50%.

The chaotic maps in combination with predictable PRNG methods such as LFSR (which is implemented by a small number of registers) are able to solve the problem of predictability by using Exclusive-OR operation between LFSR’s system and chaotic logistic map, this Theorem was proved by M. Babaei in 2011 [13].

4. Improving PRNGs

In the main part of this section we discuss about how the PRNG weaknesses could be improved. Nowadays a lot of scientists are working on this subject and could

Figure 1. Halton sequences in dimensions 1 × 20.

Figure 2. Halton sequences in dimensions 20 × 21.

Figure 3. Sobol sequences in dimensions 1 × 20.

Figure 4. Sobol sequences in dimensions 20 × 21.
prepare reliable methods to improve these defects in generators, especially the applications of PRNGs in cryptography need to be more efficient than other applications.

### 4.1. Decimation Method

In this method two PRNGs generate random numbers in two different sequences, (the type of PRNGs may be the same or different). Combination based on this method is able to generate more efficient random sequence.

Based on this method if a PRNG generated a sequence such as:

$$p_1, p_2, \ldots, p_n$$

By using of Decimation method generated a sequence like this:

$$q_1, q_2, \ldots, q_m$$

By defining that:

$$q_i = p_{jk}$$ where \( k > 1 \) chose as a constant efficient value.

Decimation algorithm is described below:

1. float* Decimation()
2. {
3. float *finalSeq;
4. boolean continueGenerating = True;
5. int count PRNG = 0;
6. char ans;
7. finalSeq = new float [100];
8. while (Continue Generating);
9. {
10. int loop Length = PRNG1();
11. for (int i = 1; i < loopLength; i++)
12. float random Number = PRNG2();
13. finalSeq [countPRNG] = random Number;
14. }
15. cout << “Do you want more? (y/n)” << endl;
16. cin >> ans;
17. if (ans == ’n’ || ans == ’N’);
18. break;
19. }
20. return finalSeq;
21. }

In other words this algorithm in Line 10 generates sequence \((p_1, p_2, \ldots, p_n)\) and in Line 12 generates sequence \((q_1, q_2, \ldots, q_m)\), so Decimation method generates the final sequence as below:

$$p_1 \cdot q_1, p_2 \cdot q_2, p_3 \cdot q_3, \ldots, p_n \cdot q_m$$

Many papers proved that generating random sequence by this method is more efficient than discrete sequences [19], meaning that:

Distribution \((\text{SeqDecimation}) > \text{distribution (PRNG1)}\)

### 4.2. XOR Operation and Combination PRNGs

One of the popular models to improve PRNG’s defects is combining \(k\) numbers of generator by Xor operation. For example, if each of the generators is defined by a primitive trinomial such as:

$$P_k^r (x) = x^4k + x^4 + 1$$

This is the main structure of Fibonacci LFSR generators which \(r_k\) is distinct primitive degrees, then proved that the combination of these \(k\) generators has a period of at least \(2^{2m-1} \prod_{k=1}^{k} (2^{2^r - 1})\).

In this case, we present various PRNGs with different efficiencies, which can be classified into three main groups.

- **Class 1**: Classic Generator XOR Classic Generator
- **Class 2**: Classic Generator XOR Modern Generator
- **Class 3**: Modern Generator XOR Modern Generator

Based on the classification classic generators (e.g. Low discrepancy methods [10], High discrepancy methods [10], LFSR methods [13] and …) and modern generators contain any type of discrete chaotic maps (e.g. Henon map [14], Logistic map [14], Gauss map [14] …).

### 4.3. Shuffling Method

In this method two PRNGs generate two different random sequences like the Decimation method, one of the sequences stores in the buffer area and the other chooses from buffer side.

Suppose that, there are two PRNGs such as:

$$P: (p_0, p_1, \ldots, p_n)$$

where \(P\) is a PRNG’s function.

$$Q: (q_0, q_1, \ldots, q_m)$$

where \(Q\) is a PRNG’s function, and \(m < n\).

By using of buffer \(B\) (minimum size \(n\), storing sequence \(P\) in the buffer \(B\) then, by using of sequence \(Q\) chooses, \(m\) values of buffer \(B\) and put them in the final result sequence.

Shuffling algorithm is described below:

1. float *Shuffling()
2. {
3. float *finalSeq;
4. float *buffer;
5. int bufferLength = 100;
6. char ans;
7. boolean continueGenerating = True;
8. int countPRNG = 0;

...
5.1. TestU01

This test is designed in four classes of modules: (a) Implementing per-programmed PRNG; (b) Implementing statistical tests; (c) Implementing per-defined batteries of tests; (d) Implementing tools for applying tests to entire family of PRNGs. These modules are implemented in the ANSI C language and offer the best collection of utilities for the empirical statistical testing [17]. The results of the test suites is classified into three classes, Small Crush (consist of 10 tests), Crush (consist of 60 tests) and Big Crush (consist of 45 tests).

In Table 1, the column $\log_2 \rho$ in the mathematical equation shows the number of period length $\rho$ in the logarithm in base 2. The column t-32 shows the CPU time which is required to generate a sample sequence with length $10^8$ of random numbers on a 32 bit computer. This computer has Intel Pentium processor of clock speed 2.8 Giga Hertz which the Ubuntu 8.10 as OS is running on it. Also the small dash (-) indicates that the test was not applied to this particular PRNG, usually because the PRNG was already failed into smaller battery. The results of TestU01 are shown in Table 1.

5.2. NIST

One of the most powerful statistical tests is NIST tests suite. It contains 15 tests which are based on null hypothesis testing. This package focuses on large types of general non-randomness on the target sequences [18]. All of the tests are standard normal and the amount of the chi-square as reference distribution. So if the current sequence which is under test is non-random, the software calculates an unacceptable value for sequence distribution. The results of eight NIST tests are shown in Tables 2-3.

<table>
<thead>
<tr>
<th>Generators</th>
<th>log$^2$</th>
<th>t-32</th>
<th>Small Crush</th>
<th>Crush</th>
<th>Big Crush</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCG$^a$</td>
<td>24</td>
<td>3.9</td>
<td>14</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LCG$^b$</td>
<td>57</td>
<td>4.2</td>
<td>1</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>LFib$^c$</td>
<td>85</td>
<td>3.8</td>
<td>2</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>MSM</td>
<td>101</td>
<td>3.0</td>
<td>5</td>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>Chaotic MSM$^d$</td>
<td>27</td>
<td>3.2</td>
<td>9</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>MPM</td>
<td>105</td>
<td>3.2</td>
<td>7</td>
<td>47</td>
<td>-</td>
</tr>
<tr>
<td>Chaotic MPM$^d$</td>
<td>29</td>
<td>3.4</td>
<td>10</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Fibonacci LFSR</td>
<td>30</td>
<td>4.1</td>
<td>17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Glaois LFSR</td>
<td>31</td>
<td>4.0</td>
<td>15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chaotic LFSR$^d$</td>
<td>32</td>
<td>4.2</td>
<td>9</td>
<td>12</td>
<td>14</td>
</tr>
</tbody>
</table>

a: ($2^{23}$, 16598013, 12820163); b: ($2^{29}$, 1313, 0); c: ($2^{31}$, 55, 24); d: logistic map.
Table 2. Results of NIST for various PRNGs (Part 1).

<table>
<thead>
<tr>
<th>Generators</th>
<th>Frequency</th>
<th>Block Frequency</th>
<th>CuSums Forward</th>
<th>CuSums Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCG(^a)</td>
<td>0.804645</td>
<td>0.764534</td>
<td>0.193567</td>
<td>0.002323</td>
</tr>
<tr>
<td>LCG(^b)</td>
<td>0.985634</td>
<td>0.893467</td>
<td>0.229087</td>
<td>0.012678</td>
</tr>
<tr>
<td>LFib(^c)</td>
<td>0.875379</td>
<td>0.026789</td>
<td>0.679834</td>
<td>0.126789</td>
</tr>
<tr>
<td>MSM</td>
<td>0.908733</td>
<td>0.128908</td>
<td>0.873456</td>
<td>0.009367</td>
</tr>
<tr>
<td>Chaotic MSM(^d)</td>
<td>0.804645</td>
<td>0.322901</td>
<td>0.265567</td>
<td>0.090388</td>
</tr>
<tr>
<td>MPM</td>
<td>0.83733</td>
<td>0.127835</td>
<td>0.783606</td>
<td>0.091678</td>
</tr>
<tr>
<td>Chaotic MPM(^d)</td>
<td>0.96372</td>
<td>0.762609</td>
<td>0.126709</td>
<td>0.201289</td>
</tr>
<tr>
<td>Fibonacci LFSR</td>
<td>0.535558</td>
<td>0.256881</td>
<td>0.125567</td>
<td>0.558502</td>
</tr>
<tr>
<td>Glaois LFSR</td>
<td>0.269087</td>
<td>0.269087</td>
<td>0.390767</td>
<td>0.389001</td>
</tr>
<tr>
<td>Chaotic LFSR(^d)</td>
<td>0.606499</td>
<td>0.483676</td>
<td>0.553505</td>
<td>0.769260</td>
</tr>
</tbody>
</table>

a: (2\(^{24}\), 16598013, 12820163); b: (2\(^{24}\), 1313 , 0); c: (2\(^{31}\), 55, 24); d: logistic map.

Table 3. Results of NIST for various PRNGs (Part 2).

<table>
<thead>
<tr>
<th>Generators</th>
<th>Rans</th>
<th>Long Run</th>
<th>Rank</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCG(^a)</td>
<td>0.876522</td>
<td>0.003634</td>
<td>0.347851</td>
<td>0.000147</td>
</tr>
<tr>
<td>LCG(^b)</td>
<td>0.753678</td>
<td>0.125620</td>
<td>0.892736</td>
<td>0.000951</td>
</tr>
<tr>
<td>LFib(^c)</td>
<td>0.595634</td>
<td>0.0913567</td>
<td>0.012673</td>
<td>0.000566</td>
</tr>
<tr>
<td>MSM</td>
<td>0.463678</td>
<td>0.001237</td>
<td>0.347851</td>
<td>0.000159</td>
</tr>
<tr>
<td>Chaotic MSM(^d)</td>
<td>0.569766</td>
<td>0.066673</td>
<td>0.248649</td>
<td>0.000159</td>
</tr>
<tr>
<td>MPM</td>
<td>0.67364</td>
<td>0.087367</td>
<td>0.001267</td>
<td>0.000159</td>
</tr>
<tr>
<td>Chaotic MPM(^d)</td>
<td>0.88383</td>
<td>0.283709</td>
<td>0.337328</td>
<td>0.000159</td>
</tr>
<tr>
<td>Fibonacci LFSR</td>
<td>0.578382</td>
<td>0.012343</td>
<td>0.859903</td>
<td>0.000159</td>
</tr>
<tr>
<td>Glaois LFSR</td>
<td>0.369001</td>
<td>0.155672</td>
<td>0.790510</td>
<td>0.000159</td>
</tr>
<tr>
<td>Chaotic LFSR(^d)</td>
<td>0.425020</td>
<td>0.174249</td>
<td>0.967341</td>
<td>0.000159</td>
</tr>
</tbody>
</table>

a: (2\(^{24}\), 16598013, 12820163); b: (2\(^{24}\), 1313 , 0); c: (2\(^{31}\), 55, 24); d: logistic map.

6. Conclusions

In this paper we discussed about some important factors to generate Pseudo-Random Numbers such as uniformity of distribution, independence, efficient length of period and unpredictability. Then we proved that chaotic logistic map is able to promote the performance of classic PRNGs which are not independent generators or do not have a long reliable period to generate random numbers. Finally the statistical tests (i.e. TestU01 and NIST suite tests) supported the main idea of the paper.

7. References


