International Journal of Control and Automation
Vol. 13, No.2, (2020), pp. 1366 - 1378

A Novel Approach to Generate Random Numbers Using Fuzzy State Automaton

*Dr. Chatrapathy K[1], Dr. Naveen Kumar B[2]
[1,2] *Department of ISE, Sahyadri College of Engineering & Management, Mangaluru, Karnataka - India*
[1]pathykc@gmail.com, [2]navkan24@gmail.com,

Abstract

A novel Pseudo Random Number Generator is developed using a Fuzzy State Automaton. The NIST STS Test Battery 800-221a has been implemented using MATLAB R.2011a. The random bit sequence generated has been tested for expected qualities of randomness, uniformity and independence using visual tests and statistical tests using NIST Test battery. Quality of the random bit sequence generated has been compared with that of true random bits HOT BITS and a standard JAVA random number generator. It is found that the random sequence developed in this work is much simple and powerful and therefore it has very good scope in future cryptographic applications.

Keywords: pseudorandom, uniformity, independence, significance, p-value, entropy

1. Introduction

In this paper, we have developed a novel Pseudo Random Number Generator (PRNG) using Fuzzy State Automaton (FSA). The random bit sequence generated has been tested for expected qualities randomness, uniformity and independence, confusion and diffusion. Algorithms are developed for implementing NIST STS 800-22 Revision 1a (2010) standard a standard test battery. The quality of PRNG-FSA bit sequence is compared with that of random number generator of Java and true random numbers obtained from HOT BITS. In section 2, basics of random number sequence and types of random number generators are discussed. Important applications of RNGs and increasing need for developing new PRNGs in various application areas are discussed in section 3. Section 4 gives basic definitions and mathematical results used in the rest of the chapter. In section 5, we have discussed random binary sequence generation using binary Finite Automaton (FA) and Fuzzy State Automaton. In section 6, we have given algorithms for implementing PRNG using FSA and NIST STS test battery. Experimentation and results are discussed in section 7. The paper ends with conclusions and future scope for research in section 8.

2. Random Number Generators

Two types of RNGs exist: a True RNG (TRNG) and a Pseudo RNG (PRNG). In TRNG random numbers are generated from the output a naturally available physical source or phenomenon. Some commonly used sources used are radioactive decay, noise of a semiconductor diode, sound samples of a noisy environment, digitized videos of a lava lamp etc. The outputs of these natural processes are truly random and therefore used to generate random numbers. TRNGs are unpredictable, slow, not scalable, often biased and expensive.

A PRNG generates a sequence of random numbers using a deterministic algorithm. It is usually implemented as a finite state machine and the initial value that is initialized with initial value called a *seed*. Once seeded, it then generates a sequence of numbers that satisfy the expected properties good

ISSN: 2005-4297 IJCA

random number sequence. Given a seed value, a PRNG always generate the same exact sequence each time it is run. Thus, the output of the PRNG is predictable. This property of predictability is most important in simulation experiments. PRNGs are easy to implement, much faster, computationally cheaper and less costly than TRNGS. The output of PRNGs should be subjected to rigorous tests to ensure various properties such as unbiasedness, independence, uniform distribution etc.

Most compilers come with a PRNG that uses a numerical algorithm or a formula to produce a sequence of numbers. Commonly available PRNGS that have been developed and studied in the past are: Linear Feedback Shift Register (LFSR), Linear Congruential Generator (LCG), Quadratic Congruential Generator 1, 2, 3, Cubic Congruential Generator II, Modular Exponentiation, Blum-Shub, Inverse Congruential Generator, Logged Fibonacci Generator and Mersenne Twister. These built-in PRNGs are not suitable for applications where high security and confidentiality are essential. Therefore numerous new and innovative PRNGs are being developed in the areas of communication and information security.

3. Applications of Random Number Generators

Random numbers play very essential roles in numerous applications in our everyday life. Random numbers with high-quality are very crucial in the areas such as Computer Science, Mathematics, VLSI Circuit testing, Computer Games, Cryptography, Monte Carlo Simulation, Information security, Brownian dynamics, Stochastic Optimization, Communication Protocols, Molecular dynamics, Robotics and many other areas of human life. Advancements in high speed computing, internet computing, wireless networks, telecommunications etc. have resulted in explosive growth of research activities which along with various commercial applications demand a very high speed and high quality random number generator. PRNGs with acceptable qualities are very crucial in these applications. Production of PRNG with acceptable quality is a very challenging task.In the current digital era random numbers are used in e-mail access, cashless transactions, point of sale, net-banking, prepaid cards, ATMs, internet trade, prepaid cards, wireless keys, online reservation systems, general cyber-security and many more daily activities. In *Cryptography* random numbers are extensively used for key generation, authentication protocols, protection against side-channel attacks and encryption systems. For applications such as *stochastic simulation*, the masking of protocols and online gambling, huge amount of random numbers are needed and thus fast PRNGs are required. In *computer and information security*, random numbers are used for authentication, protecting the integrity, confidentiality and authenticity of information resources.

4. Basic Concepts

Definition 1: A Random Bit Generator (RBG) is an algorithm or a device which generates a sequence of independent, unpredictable and uniformly distributed sequence of bits.

Definition 2: A Pseudo Random Bit Generator (PRBG) is an algorithm that takes a binary sequence of small length as input and outputs a very long a binary sequence that looks random.

Definition 3: A PRNG is an algorithm that generates longer string that looks random from a short string called seed.

Definition 4: (Uniform distribution) If N observations in the interval [0, 1] are divided into *n* subintervals of equal length, then the expected number of observations in each subinterval is [*N/n*].

Definition 5: (Independence) A property of random number sequence in which a next value is generated is independent of the previous values in the sequence.

Definition 6: A *run* of a random bit sequence is a continuous string of 1s preceded and ended by 0s and *gap* is a continuous string of 0s preceded and ended by 1s.

Definition 7: *Level of significance* (α) is a probability that a bit sequence ω appears non-random even when it is generated by a good random number generator.

Normally the α value is in the range [0.001, 0.01] and set prior to a test. When $\alpha = 0.001$, it means one sequence out of 1000 sequences generated does not pass the randomness test.

Definition 8: A *p-value* is the probability that random sequence generated is less than the quality of perfect random number sequence.
In each NIST test, a p-value is computed for each bit sequence using predetermined test statistic. The sequence is perfectly random when p = 1and completely nonrandom when p = 0. The sequence is considered random if $p \geq \alpha$ and nonrandom otherwise. That means some non-randomness present in the sequence when $p < \alpha$. In the present work we have conducted all the tests using $\alpha = 0.001$.

Definition 9: (**erfc**) The *complementary error function* related to normal cdf is defined as

$$\mathbf{erfc(z)} = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} \mathbf{e}^{-u^2} \mathbf{du}$$

Definition 10: (igamc) The incomplete gamma function Q (a, x) is defined as

$$\mathbf{Q(a,x)} = \frac{1}{\Gamma(a)} \int_{z}^{\infty} \mathbf{e}^{-t}\ \mathbf{t}^{a-1}\ \mathbf{dt}$$

Definition 11: Standard Normal Cumulative Distribution Function

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{z}^{\infty} \mathbf{e}^{\frac{-u^2}{2}}\ \mathbf{dt}$$

Definition 12: Chi^2 statistic

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

Here, O_i and E_i denote observed and expected frequencies of the test measure respectively

5. PRNG Using Fuzzy State Automaton (PRNG-FSA)

The objective of our present work is to develop (design, implement and test) a novel PRNG using Finite Automaton (FA). The work is mainly motivated from the paper [1].

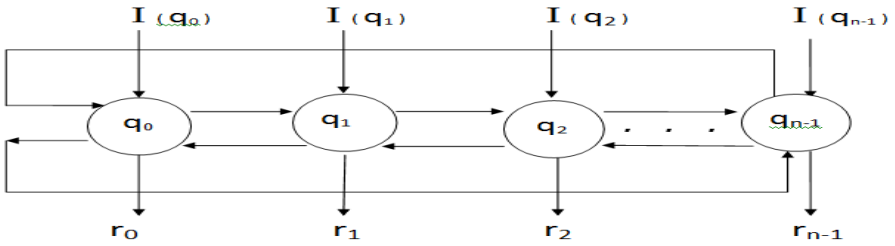

Fig.1. FA to generate Random Bit Sequence

We first discuss the construction and working of FSA (Fig. 1) and then we will consider designing FSA that will be used to generate random bit sequences.

5.1 Random Bit Sequence Generation Using FSA

We first consider the Finite Automaton (FA) M shown in Fig 1 above. At any time t, each state q_i will have a state value 0 or 1 (binary). Initially, when started at time t_0, M is loaded with binary values $I(q_0)$, $I(q_1)$, … $I(q_{n-1})$. M is considered to be circular. Let $r_0(t)$, $r_1(t)$, $r_2(t)$ … $r_{n-1}(t)$ denote the current values of states q_0, q_1 … q_{n-1} respectively at time t. The *configuration* (global state) of M is a sequence of state values at any time t denoted as $C(t) = (r_0(t), r_1(t), r_2(t), \ldots r_{n-1}(t))$ where $r_i(t) = 0$ or $r_i(t) = 1$ for $0 \leq i \leq (n-1)$. Automaton M runs continuously required number of times producing the output bit sequence $r_0r_1r_2\ldots r_{n-1}$. At each time step t+1, values of all states are updated synchronously using a Boolean function 'f' of 3 variables

$$r_i(t+1) = f(r_{i-1}(t), r_i(t), r_{i+1}(t)) \quad (1)$$

The function f is a next state function (nsf) or state update function. In the above diagram, the next state value of the state q_i is a function of the current values of states (q_{i-1}, q_i, q_{i+1}). In general, if the number of neighbors is k, the next value of state q_i is calculated by

$$r_i(t+1) = f(r_{i-k}(t), r_{i-k-1}(t), \ldots, r_{i-1}(t), r_i(t), r_{i+1}(t), r_{i+2}(t), r_{i+3}(t), \ldots, r_{i+k}(t)).$$

Thus for k neighbors, state update function f is a function of 2k+1 binary values. Thus 2^m different state update functions are possible, where $m = 2^{2k+1}$. For example, when k = 1 we have the following possibilities.

$x_{i-1}(t)\, x_i(t)\, x_{i+1}(t)$:	111	110	101	100	011	010	001	000	
$x_i(t+1)$:	0	0	0	0	0	0	0	0	(rule-0)
$x_i(t+1)$:	0	0	0	0	0	0	0	1	(rule-1)
					.				
					.				
$x_i(t+1)$:	1	1	1	1	1	1	1	1	(rule-255)

Therefore, when k = 1 we have $2^8 = 256$ different update functions.). Each of these rules can be expressed in CNF form using Boolean operators (NOT, AND, OR). For example, rule-30 is expressed as

$$r_i(t+1) = f(r_{i-1}(t), r_i(t), r_{i+1}(t)) = r_{i-1}(t) \text{ XOR } (r_i(t) \text{ OR } r_{i+1}(t)) \quad (2)$$

Random bit sequences are generated using FSA in two ways: serial and parallel. In serial method, a single automaton with large number of states is run very large number of times and one-bit output (usually from the middle state) is sampled as a random bit. In parallel method, hundreds of binary FSA, initially loaded with different patterns (seeds) are run in parallel and one-bit from a selected state from each automaton is sampled as output and these bits are padded. In both cases, randomness of the output sequence generated depends on the seed pattern, number of neighbors and state update function.

5.2 Random Bit Sequence Generation Using Fuzzy State Automaton

Fuzzy State Automaton(FSA) used in our work is designed using the following steps:

1. Design a k-neighbor binary FA with n-states as discussed in the previous section.
2. Express the state update function used in binary FSA in CNF form using basic boolean operators (NOT, AND, OR).
3. A fuzzy equivalent of f is obtained by replacing AND, OR and NOT operators in f by fuzzy t-norm, s-norm and complement operators respectively.
4. When started at time t_0, load the states q_0, q_1, … , q_{n-1} of FSA with initial fuzzy values $I(q_0)$, $I(q_1)$, …, $I(q_{n-1})$ respectively.

For example, a fuzzy state update function f for a 1-neighbor FSA using rule-30 mentioned in (1) can be implemented using fuzzy operators as follows:

$r_i(t+1) = f(r_{i-1}(t), r_i(t), r_{i+1}(t)) = r_{i-1}(t)$ XOR $(r_i(t)$ OR $r_{i+1}(t))$.
Let $a = r_{i-1}(t)$, $b = r_i(t)$ and $c = r_{i+1}(t)$
$d = [b \vee c]$
Therefore we have
$r_i(t+1) = a$ XOR d
$= [a \wedge d'] \vee [a' \wedge d]$
$= [a \wedge (1 - d)] \vee [(1 - a) \wedge d]$ (3)

In equation (3) the operators $\vee$ and $\wedge$ denote the fuzzy OR (s-norm) and fuzzy AND (t-norm) operators respectively. Some commonly used t-norm and s-norm operators are given table 1 below.

class of fuzzy operators	**t-norm $(a \wedge b)$**	**s-norm $(a \vee b)$**	**complement (a')**
Łukasiewicz	Max (0, a + b - 1)	Min (1, a + b)	1 - a
Einstein	(ab) / (2 – [a + b - ab])	(a + b) / (1+ ab)	1 - a
Hamacher	(ab) / (a + b – ab)	(a + b – 2 ab) / (1 – ab)	1 - a
Dombi $[\lambda \in (0, \infty)]$	$\frac{1}{1+[(1/a-1)^{-\lambda} + (1/b-1)^{-\lambda}]^{(1/\lambda)}}$	$\frac{1}{1 + [(1/a-1)^{\lambda} + (1/b-1)^{\lambda}]^{(1/\lambda)}}$	$\frac{(1 - a)}{(1 + \lambda.a)}$
Yager $[\omega \in (0, \infty)]$	Min $(1, [a^{\omega} + b^{\omega}]^{1/\omega})$	Min $(1, [(1-a)^{\omega} + (1-b)^{\omega}]^{1/\omega})$	$(1 - a^{\omega})^{1/\omega}$

Table 1. Commonly used fuzzy AND, OR and NOT operators

The randomness of the bit pattern generated depends on various factors such as number of states, initial global state (I), number of warm-up cycles, fuzzy operators selected and value of the parameter (λ, ω) .

The *Fuzzy State Automaton* used in this work is defined as follows.

Definition 13: A Fuzzy State Automaton (FSA) is a 3-tuple M = (Q, I, f) where Q is a finite set of states, I: $Q \rightarrow [0, 1]$ is a fuzzy initial state and f: $Q \times [0, 1]^3 \times Q \rightarrow [0, 1]$ is a state update function.

Here I is the set of initial seed values for all states.

The FSA is shown pictorially in Fig.1 above. Working of FSA is similar to that of binary FSA. But, state values and operators used in state update functions are fuzzy. In the proposed model, each state q_i has a fuzzy value $r_i \in [0, 1]$. The sequence of fuzzy values of all states at any instance of time t denoted as $C(t) = (r_0(t), r_1(t), \ldots, r_n(t))$ is called as *fuzzy state* of FSA at time t. The initial state of FSA denoted $C(t_0) = (I(q_0), I(q_1), \ldots, I(q_{n-1}))$. The FSA moves through the states $C(t_1)$, $C(t_2)$ … at time instances t_1 , t_2 , … respectively. The next state value of each state q_j is computed using a fuzzy function of current state value of q_j and its neighboring states. When moving from one configuration to the next, all the states are updating state values synchronously. Let $C_t = (r_0(t), r_1(t), \ldots, r_{n-1}(t))$ be a current state of FSA at time t. The next state of the FSA at time (t+1) denoted $C(t+1) = (r_0(t+1), r_1(t+1), \ldots, r_{n-1}(t+1))$ is computed as follows.

$r_0(t+1) = f(r_{n-1}(t), r_0(t), r_1(t))$

$r_1(t+1) = f(r_0(t), r_1(t), r_2(t))$

$r_2(t+1) = f(r_1(t), r_2(t), r_3(t))$

. . . . . .

$r_{n-1}(t+1) = f(r_{n-2}(t), r_{n-1}(t), r_0(t))$

From each fuzzy state C (t_1), C (t_2), … C (t_m) of FSA, the output bit sequence is generated by sampling selected bits from the binary representation of each state value r_i. The bit sequences generated from each fuzzy state are padded together to form the final output bit sequence. Here m is the number of times we run the FSA to generate random bit sequence.

Example output using rule-30

Enter no of states: 4

Initial fuzzy state (I): [0.852957 0.660398 0.953486 0.715240]

Enter no of cycles:3

==> Iteration 1

Fuzzy state 1: [0.284756 0.539602 0.484756 0.284760]

Binary:[01001000101001010100010000000101001001001001010101010
101101001010000000101010000000100010101000100010010100
010110001010010101000100000001010010010010010101000010
010010001010010000001000100000100100100100010000101010]

Pattern Extracted: 1000101 0100101 1000101 1000101 (bits 5-11 of each state)

Sequence generated: 10001010100101100010111000101

==> Iteration 2

Fuzzy state 2: [0.389212 0.439712 0.379612 0.892536]

Binary:[01010100101000001000101000101000100000001010010100001000
010101001010000010001010000101001010000101000100010101010
010101001010000010001010001010001000000010100100101010101
101010101010001010010101010101010001010010010000100101010]

Pattern Extracted: 0100101 0100101 0100101 1010101 (bits 5-11of each state)

Sequence generated: 01001010100101010010111010101

==> Iteration 3

Fuzzy state 3: [0.345000 0.398159 0.475000 0.645000]

Binary:[01010000010100010100010000100101001010101001010100101010
010100010010000010100000000101010100001001010001000101010
010100000101000101000100001001010010101010010101001010101
110100000101000101000100001010010010010100010010101001010]

Pattern Extracted: 0000010 0001001 0000010 0000010 (bits 5-11 of each state)

Sequence generated: 00000100001001000000100000010

Output bit sequence:
10001010100101100010110001010100101010010101001011010101000001000010010000010000000010

6. Implementation

The application developed has two major modules: *Generator module* and *NIST STS Test module*. The Generator module contains algorithms for PRNG using FSA and associated functions to implement state update function using fuzzy AND, OR and NOT operators. NIST STS Test module contains functions for 15 NIST tests specified in NIST 800-22.a (2010) standard test suite. All algorithms are implemented using MATLAB R2011.1a.

6.1 Generator Module (Implementation of PRNG using FSA)

In this section we give algorithms used to implement a PRNG using FSA.
Fuzzy Operators used:

AND (∧) : $$t_\lambda(a,b) = \frac{1}{1+\left[\left(\frac{1}{a}-1\right)^\lambda + \left(\frac{1}{b}-1\right)^\lambda\right]^{\frac{1}{\lambda}}}$$

OR (∨) : $$s_\lambda(a,b) = \frac{1}{1+\left[\left(\frac{1}{a}-1\right)^{-\lambda} + \left(\frac{1}{b}-1\right)^{-\lambda}\right]^{\frac{1}{\lambda}}}$$

NOT (a) : $$\bar{a} = \frac{1-a}{1+\lambda a}$$

Here a, b ∈ [0, 1] are fuzzy values and λ ∈ (0, ∞) is the parameter to compute the values of AND, OR and NOT operations.

State update function used:

$r_i(t+1) = f(r_{i-1}(t), r_i(t), r_{i+1}(t)) = r_{i-1}(t)$ XOR ($r_i(t)$ OR $r_{i+1}(t)$) (*rule-30*)

Function PRNG ( ). This function simulates n-state FSA as discussed in section 7.5.2. Initial configuration I is obtained using standard random number generator (rand in MATLAB) and *m* denotes the number of repetition steps used to generate random bit sequence. The bit sequence generated during each step is stored in OUTFILE-1 for further statistical testing. Next state function nsf is called to update the configuration of FSA during each iteration.

1. OUTFILE-1 = 'E:\KCPRNG\data.txt'
   fid = fopen (OUTFILE-1, 'w') {open output file in write mode}
2. Read n {number of states}
   Read m {number of repetitions}
3. {Initialize FSA with initial fuzzy state I}
   a = rand (1, a) {a is n-element array to store the configuration of FSA}
4. {Run FSA m-times}
   for i = 1 to m
      b = **nsf** (a) {move FSA to next configuration using fuzzy next state function}
      a = b {update the states values synchronously}
      {Generate bit sequence}
      bin = [ ] {array to store bit sequence during each step}
      for j = 1 to n
         t = binary (a [i]) {get binary equivalent of state value of q_i}
         p = t [5 .. 11] {extract bit_5 .. bit_{11} (6-bits) discarding first 4-bits}
         bin = [bin pat] {append sampled bits}

```
            end
            write (fid, bin) {write a bit sequence generated to the file}
        end
5.  fclose (fid) {close the output file}
6.  end.
```

Function nsf (x). This function takes an array x which is a current fuzzy state (configuration) of FSA and updates the state values of each state q_i synchronously by applying fuzzy function *update* to each state. Variables a, b, c represent the state values of left neighbor, state q_i and right neighbor respectively. The updated fuzzy state t is returned as output.

```
1.  n = | x |
2.  t = zeros(1, n) {n element array to store updated state values}
3.  {update current value of each state and store in the array t}
    for  i = 1 to n
        if  (i =1)
            a = x [n]       {state value of left neighbor of q_i}
        else
            a = x [i-1]
        endif
        if  (i = n)
            c = x [1]       {state value of right neighbor of q_i}
        else
            c = x [i+1]
        endif
        b = x[i]  {state value of q_i}
        t[i] = update (a, b, c) {next value of state q_i}
    end
4.  return t
```

Function update (a, b, c). This function takes current state values of states (left neighbor, q_i, right neighbor) and computes the next state value of q_i by applying fuzzy state update rule.

```
1.  d = b OR c
2.  t =  (a AND (NOT(d)) OR (NOT(a) AND d)        {r_i (t) = [a ∧ (1-d)] ∨ [ (1-a) ∧ d]}
3.  return t
```

Function OR (a, b). This function computes the value of s_λ (a, b)

```
1.  t1 = (1 / a - 1) ^ (- λ) + (1 / b - 1) ^ (- λ)
2.  t2 = 1 + t1 ^ (1 / λ)
3.  t = 1.0 / t2
4.  return t
```

Function AND (a , b). This function computes the value of t_λ (a, b)

```
1.  t1= (1 / a - 1) ^ λ  +  (1 / b - 1) ^ λ)
2.  t2 = 1 + t1 ^ (1 / λ)
3.  t = 1 / t2
4.  return t
```

Function NOT (a). This function computes the value of $\bar{a}$

```
1.  t = (1 – a) / (1 + λ a)
2.  return t
```

6.2 Implementation of NIST STS Test Battery

We have developed algorithms for implementing a battery of 15 different tests specified by NIST STS 800-22 Revision 1a (2010) standard. Algorithms have been implemented using

MATLAB R2011a.

7. Experimentation and Results

The experiment conducted has three stages

1. Generation of random bit sequence
2. Testing of random bit sequence
3. Comparison of results with standard random numbers

7.1 Generation of Random Bit Sequence

A 32-state FSA is implemented with the initial fuzzy state I (32 - element array initialized with 32 random numbers generated using MATLAB). The FSA is run for 5000 warm-up steps and then run for another 1000 steps to generate 1000 sequences. In each step, value of each state (fuzzy value) is converted into binary equivalent and then 4-bits (bits 5-8) are sampled as output. Thus, 128-bits (32 × 4) are generated in each step and stored in output file (OUTFILE-1). The bit sequence generated is tested quickly using *visual tests* to ensure the basic properties of randomness, uniformity and independence. The experiment is repeated with different values of λ and the sequences produced are tested quickly using visual tests. The resulting sequence is further subjected to rigorous statistical testing using NIST STS test battery.

7.2 Testing of Random Bit Sequence

A random number sequence should have the acceptable qualities of uniform distribution, independence and randomness. In addition, a cryptographically robust random number sequence should also have the properties of confusion, diffusion, avalanche effect and completeness. The output binary sequence generated from PRNG-FSA is tested in two stages.

1. Visual Tests
2. Statistical Testing Using NIST Test Battery

7.2.1 Visual Tests

In the first stage of testing, we generated 5000 random integers in the range 100-250 using FSA and conducted the following visual tests to quickly identify the irregularities such as patterns, bias, outliers and relationships between numbers that may be present in the sequence.

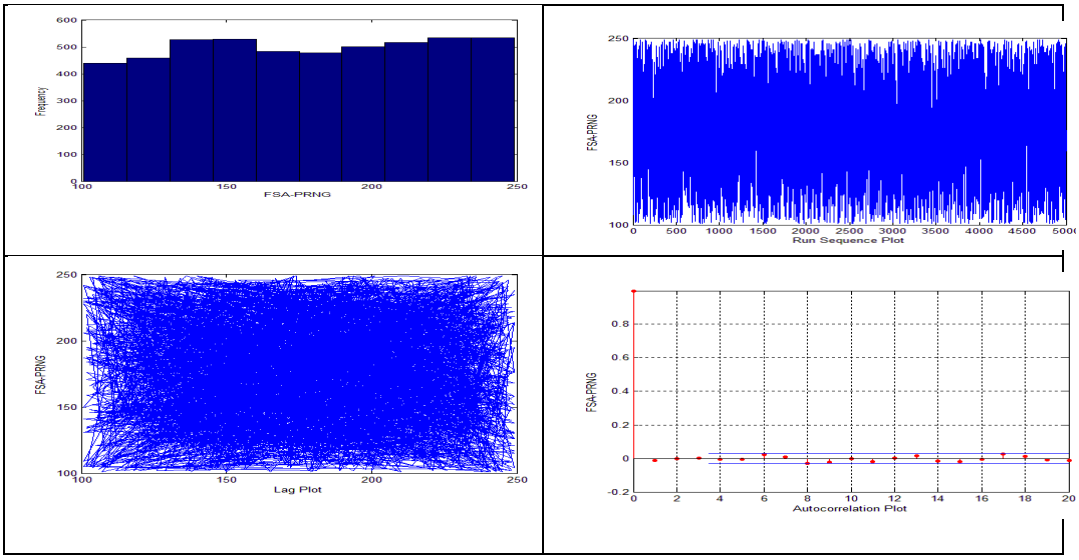

Fig. 2. Results of Visual Tests

From the visual tests, it is obvious that, the random bit sequence generated from PRNG-FSA is *mostly random*. The sequence is tested further more rigorously using NIST STS battery.

7.2.2 Statistical Testing Using NIST Test Battery

We have implemented a test battery NIST STS 800-22.1a (2010) Standard containing a series of 15 tests using MATLAB R2011.1a. The complete description of NIST tests and interpretation of test results is given in [6]. Following parameters are used by NIST tests.

n denotes the length of the sequence to be tested
M denotes the length of the block
m denotes the length of the template to be matched
R denotes the no of rows and C denotes the no of columns
B is the number template to be matched
L denotes the length of sub block
Q denotes the no of initialization blocks
K denotes the number of test blocks.

The values of test parameters used when conducting NIST tests on PRNG-FSA random number sequence are shown in table 2.

Test#	Test	Parameters
1	Frequency (monobit) Test	n = 100
2	Frequency (block) Test	n = 100 , M = 10
3	Runs Test	n = 100, $\tau=2/\sqrt{n}=0.2$, n=100
4	Longest Runs Test	n =100, M=8
5	Binary Matrix Rank Test	N= 38912, R=32, C=32
6	Spectral Test	n = 1000
7	Non-overlapping Template Matching Test	n =100, template B, m=$\lvert B \rvert$ = 9
8	Overlapping Template Matching Test	n = 10^6, template B, m=$\lvert B \rvert$ = 9
9	Maurer's universal Test	n = 10^6, L=7, Q =128, K=1485
10	Compression Test	n = 10^6, M =100
11	Serial Test	n = 10^6, M=3
12	Approximate Entropy Test	n = 100, M=3
13	Cumulative sum Test	n = 100
14	Random Excursions Test	n = 10^6
15	Random Excursions Variant Test	n = 10^6

Table 2. Parameters used in NIST Tests

Data set used for Conducting NIST tests:

Sequence-1: The output binary sequence generated using PRNG-FSA containing 1000 lines each line containing 128 bits (outfile1.txt).

Tests are conducted using the significance level $\alpha = 0.001$ which indicates that one sequence out of 1000 sequences is rejected by the test due to the presence of some kind of randomness in the sequence. In each test a p-value for each of the 1000 sequences is calculated and average p-value for entire data set is calculated. The results of NIST tests on the random number sequence generated by PRNG-FSA are summarized in the Table 3 given below. The results in table 3 are depicted graphically in Fig. 3 and Fig.4 shown below. It is obvious from the results shown in the table 3 and the graphs in Fig.3 and Fig.4, the pass percentages of NIST tests are almost uniformly distributed with the values

ranging from 85.9% to 100%. The pass percentage of Frequency Tests is 87.4% which indicates that 870 out of 1000 binary sequences have passed the randomness tests.

Test No	Test Name	Pass %	AVG P-value	Result
1	Frequency (Monobit) Test	87.40	0.335	Pass
2	Frequency (Block) Test	98.70	0.601	Pass
3	Run Test	98.60	0.486	Pass
4	Longest Run Test	98.90	0.611	Pass
5	Binary Matrix Rank Test	92.00	0.472	Pass
6	Spectral Test	100.00	0.259	Pass
7	Non-overlapping Template Match Test	100.00	0.897	Pass
8	Overlapping Template Match Test	89.60	0.520	Pass
9	Universal Test	88.70	0.484	Pass
10	Compression Test	100.00	0.254	Pass
11	Serial Test	97.50	0.719	Pass
12	Approximate Entropy Test	94.10	0.351	Pass
13	Cumulative Sum Test	97.52	0.506	Pass
14	Random Excursion Test	94.10	0.351	Pass
15	Random Excursion Variant Test	85.90	0.490	Pass

Table 3. NIST Test Results on PRNG-FSA Sequence (Level of significance α = **0.001**)

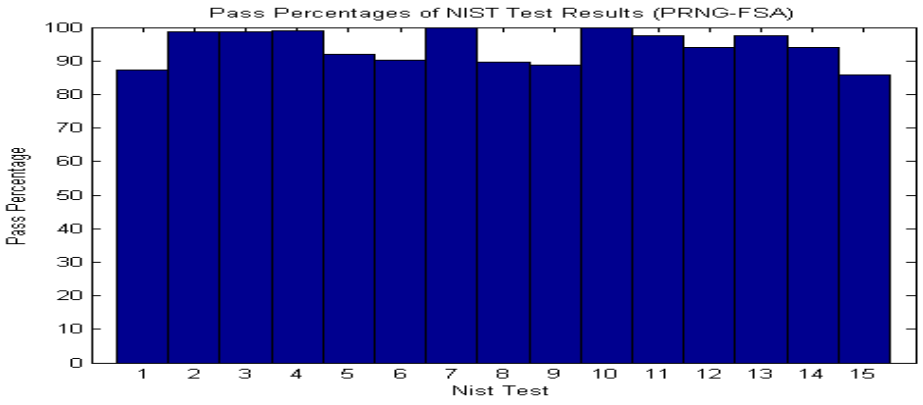

Fig.3. Pass Percentages of NIST Tests (PRNG-FSA)

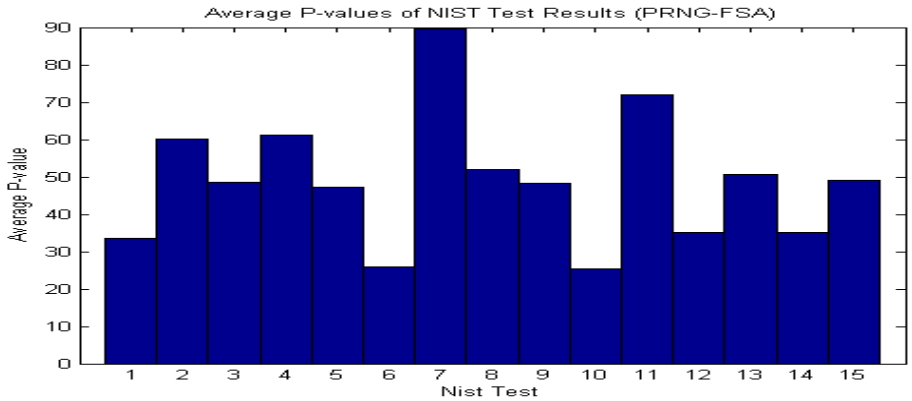

Fig. 4 Average P-Values of NIST Tests (PRNG-FSA)

7.3 Comparison of Results with Standard Random Numbers

Two standard random bit sequences are generated to compare the quality of the random bit sequence generated using PRNG-FSA.

1. **Sequence-2** (OUTFILE-2): 2000 lines of bit sequence each line containing minimum 128 bits are generated using standard random number generator available in Java.
2. **Sequence-3** (OUTFILE-3): 4000 lines of *true random bit sequences* each line with 240 bits obtained from HOTBITS.

Above two standard random bit sequences are tested using NIST STS test battery separately and the results are summarized in Table 4 given below.

Test No.	Test Name	BIT SEQUENCE					
		PRNG-FSA		Java Standard		HOTBITS	
		Pass%	Average	Pass%	Average	Pass%	Average
1	Frequency (Monobit) Test	87.40	0.335	90.17	0.335	75.19	0.335
2	Frequency (Block) Test	98.70	0.601	98.34	0.603	99.32	0.574
3	Run Test	98.60	0.486	99.09	0.493	98.29	0.464
4	Longest Run Test	98.90	0.611	99.24	0.642	83.82	0.289
5	Binary Matrix Rank Test	92.00	0.472	94.86	0.513	90.06	0.444
6	Spectral Test	90.20	0.259	92.73	0.281	95.81	0.028
7	Non-overlapping Template	100.00	0.897	100.0	0.899	100.0	0.964
8	Overlapping Template Match	89.60	0.520	90.47	0.548	93.71	0.414
9	Universal Test	88.70	0.484	84.57	0.511	94.35	0.495
10	Compression Test	100.00	0.254	100.0	0.299	100.0	0.655
11	Serial Test	97.50	0.719	98.03	0.723	97.71	0.698
12	Approximate Entropy Test	94.10	0.351	95.61	0.367	95.42	0.350
13	Cumulative Sum Test	97.52	0.506	100.0	0.494	100	0.491
14	Random Excursion Test	94.10	0.351	95.61	0.367	95.42	0.350
15	Random Excursion Variant	85.90	0.490	83.66	0.482	62.67	0.357

Table 4. Comparison of Random Bit Sequences (Level of significance $\alpha = 0.001$)

We compare the quality of the three random sequences considered in this work namely PRNG-FSA sequence, Java sequence and Hotbits sequence using pass percentages and average p-values of the NIST tests conducted on them. The results of comparison are shown graphically in Fig. 5 and Fig. 6 given below.

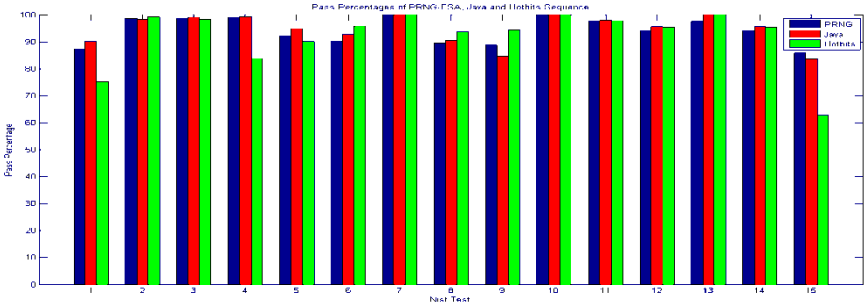

Fig. 5 Comparison of Pass Percentages

From the above graph, we conclude that the pass percentage of PRNG-FSA is almost

comparable to that of Java sequence and much better than the pass percentage of Hotbits sequence.

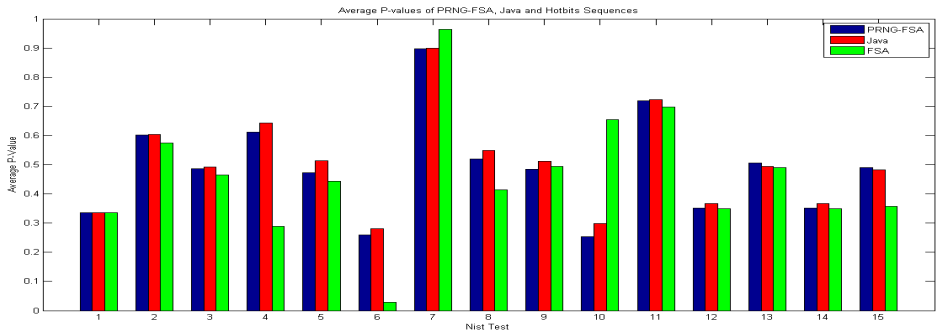

Fig. 6 Comparison of Average P-values

The graph in Fig.5 indicates that average p-values of PRNG-FSA and Java sequence are almost comparable in all 15 tests whereas the average p-values of PRNG-FSA are higher than that of Hotbits except in tests 7 and 10. This indicates that quality of PRNG-FSA bit sequence is better than that of a true random sequence HOTBITS.

8. Conclusions

A PRNG-FSA with 32-states and rule-30 has been implemented and tested for acceptable qualities using visual tests and NIST STS that has been implemented in MATLAB. The results obtained have been compared with true random bit sequence HOT BITS and a standard JAVA random number generator. It is observed that average p-values of PRNG-FSA and Java sequence are almost comparable in all 15 tests whereas the average p-values of PRNG-FSA are higher than that of Hotbits except in tests 7 and 10. It can be observed that infinite possibilities exists to experiment the generator by varying number of states, state update rule, initial seed and neighborhood states. The PRNG-FSA is much flexible and a slight change in aspect of the PRNG design leads to very large variations in the bit sequences generated. It is obvious that the PRNG developed in this work is much simple and powerful and therefore it has very good scope in future cryptographic applications.

References

[1] Wolfram, "*Random Sequence Generation by Cellular Automata*", Advances in Applied Mathematics 7, PP 123-169, 1986.

[2] David H. K. Hoe et.al. "*Cellular Automata-Based Parallel Random Number Generators Using FPGAs*", Hindawi Publishing Corporation, International Journal of Reconfigurable Computing, 2012.

[3] Brent, "*Fast and Reliable Random Number Generators for Scientific Computing*", Oxford University Computing Laboratory, 2004.

[4] Belfedhal A.E Faraoun K.M. "*Fast and Efficient Design of a PCA-Based Hash Function*", International Journal of Computer Network and Information Security, 6, PP 31-38, 2015.

[5] Madhuri.A, "*Hybrid Cellular Automata-Based Pseudo Random Sequence Generator for BIST Implementation*", International Journal of Research Studies in Science, Engineering and Technology, Volume 2, Issue 9, PP 72-76, 2015.

[6] Rukhin et.al. "*A Statistical Test Suits for Random and Pseudo Random Number Generators For Cryptographic Applications*", NIST special publication 800-22, 2001

ISSN: 2005-4297 IJCA