Fuzz-Fortuna: A fuzzified approach to generation of cryptographically secure Pseudo-random numbers

M. Ali Akbar and M. Zulkifl Khalid
College of Electrical and Mechanical Engineering
National University of Sciences and Technology (NUST)
Peshawar Road, Rawalpindi, Pakistan
Email: {ali.akbar.ceme, zulkifl}@gmail.com

***Abstract*— A fuzzy based adaptive algorithm for the reseeding operation of Fortuna is presented. Fortuna is a pseudo-random number generation algorithm, originally suggested by Ferguson and Schneier[1]. The algorithm is specifically designed to be cryptographically secure from known attacks. However the described algorithm suffers from a lack of an algorithm which could adapt the rate of reseeding according to variations in the amount of truly random data being gathered from the environment at any time. The Fortuna algorithm performs the reseeding action after a fixed number of iterations. This paper presents concept as well as software implementation of a novel technique using *fuzzy* approach to tackle this problem. The resulting algorithm has been named Fuzz-Fortuna. Fuzz-Fortuna has been tested using various techniques and has shown considerable improvement in results as compared to the ordinary Fortuna algorithm.**

I. Introduction

A. Background

A pseudorandom number generator (PRNG) is an algorithm that follows a deterministic procedure to generate a sequence of numbers that approximate 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 the PRNG's state. In contrast a true random number generator (TRNG) provides truly random data gathered from undeterministic phenomenon occuring in nature.

For cryptographic purposes, random data is required for generation of keys etc. Truly random data, though ideal for this purpose, is usually not available at the fast data rate required. To overcome this problem, many pseudorandom number generator algorithms have been developed which make periodic use of truly random data available at low rate to make their current state undeterministic. These algorithms usually produce random numbers of sufficient quality to be used in practical cryptographic scenarios.

Most of these cryptographically secure PRNGs use algorithms which are very hard to be reverse engineered. That is, the knowledge of a sufficiently large number of consecutive outputs should not reveal the current state of the generator. The algorithms ideal for this purpose are hashing and encryption algorithms e.g. Data Encryption Standard (DES) or Advanced Encryption Standard (AES). Fortuna is one such algorithm based on hashing and encryption algorithm. It uses SHA for hashing and AES for encryption purposes.

B. Problem Identification

Fortuna algorithm uses truly random data gathered from entropy sources to make its state undeterministic. This process is known as 'reseeding' operation. However, the fortuna algorithm performs the reseeding operation after a fixed number of iterations. Tests show that the quality of random numbers generated is increased significantly with increase in frequency of reseeding operation provided enough random data is available from entropy sources. This indicates the need of an algorithm which adopts the frequency of reseeding operation according to the amount of data being gathered from the environment.

This paper presents a modified version of Fortuna algorithm namely Fuzz-Fortuna. First of all we describe the related work. Then, the working of Fuzz-Fortuna is explained in detail. It is followed by a detailed analysis and results of tests done on the output of the developed system. The results prove our assertion about the robustness and improved performance of our proposed technique.

II. Related Work

A lot of work has been done in the field of random number generation. The proposed algorithms and implementations are too many to list. The most simple and easy to implement PRNGs are based on linear shift registers. Examples of such PRNGs are the Lagged Fibonacci Generator[2] (used in MATLAB) and Blum Blum Shub[3]. [4] describes the operation of Mersenne Twister(MT) RNG. MT is a 623 dimensionally equidistributed PRNG. An improved version of MT is the SIMD-oriented Fast Mersenne Twister algorithm given in [5].

The number of algorithms specifically designed for generation of cryptographically secure pseudorandom numbers is relatively few. An important PRNG employing hashing and encryption techniques is the Yarrow algorithm [6]. This work is based on the software and hardware implementation of Fortuna algorithm by [7]. In that work, the authors have shown results which prove the effectiveness of the Fortuna algorithm. However, the work lacks an adaptive algorithm for the process of reseeding in the generator core. Our work is an effort towards solving this problem.

III. The Fuzz-Fortuna Model

Fuzz-Fortuna comprises of four parts: an entropy accumulator, a generator core, a system for seed file management and a fuzzy inference system for controlling the frequency of reseeding operation. The first three parts are same as the original fortuna model described in detail in [7]. However, they are briefly described here for continuity. The fourth part, Fuzzy Inference System is the new proposed addition which implements the adaptive algorithm for controlling of reseed operation.

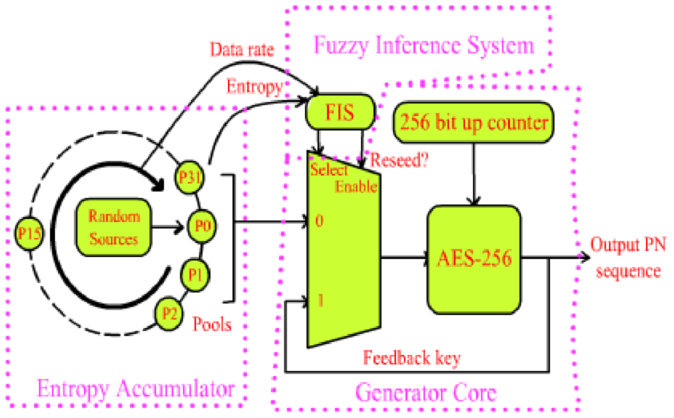


Fig. 1. Fuzz-Fortuna Model

A. Entropy Accumulator

The purpose of an accumulator is to store the truly random data obtained from the entropy sources and use it in reseeding operation to make the state of the system undeterministic. The security of the system lies on the observation that the system will remain secure (unpredictable by an attacker) as long as at least one active source of entropy is not controlled by the attacker.

Pools are used for storing the random data accumulated. Data gathered from random events from the entropy sources is uniformly and cyclically distributed amongst 32 pools, labeled P_0, P_1, ..., P_{31} respectively. To store data of variable length in the pool of constant size, we use a special hashing algorithm known as SHA-256 hash function[8], thereby maintaining a constant pool size of 32 bytes. When pools have accumulated enough random data, the generator can be reseeded.

The process of choosing pools to provide data for reseed is as follows. A counter $data\_run$ tracks the number of times the generator has been reseeded from the pools. This counter determines which pools will be used in the current reseed, i.e. pool P_i will be included in the reseed if $i+1$ divides $data\_run$. Hence, P_0 is included in every reseed, P_1 in every second reseed, and so on. This results in some pools being less used in reseed, thus storing greater entropy.

B. Generator

The generator consists of a block cipher (AES-256 [9]) used for encryption. The block cipher takes a 256 bit up-counter as input, a 256 bit random key from accumulator and produces a 256 bit PN sequence at its output. The output of generator is also fed back as new random key if reseeding needs to be done but the accumulator pools do not have enough random data for reseeding operation. The data would have a period of 2^{256} if reseeding is done after the counter has overflowed. Since periodic data is not secure, the generated data is limited by a maximum limit of key oldness using the fuzzy inference system. The generator changes the key or reseeds from the pools if the fuzzy inference system (FIS) determines that the key is too old or the entropy level of pools is quite high.

C. Seed File Manager

To make sure, that the generator always starts from an unpredictable state, a seed file is used to reseed the generator. The system periodically saves the seed file, thus maintaining it in an unpredictable state for future use. The seed file manager takes care of reading from seed file, writing to seed file and saving the seed file.

D. Fuzzy Inference System

A novel fuzzy algorithm has been developed in this paper which provides an adaptive method of reseeding the generator core according to the random data supplied by the environment. The fuzzy model was built and tested using MATLAB. The generated FIS was exported as .fis file and used in the software implementation using the header files provided by MathWorks. The details of the fuzzy inference system (FIS) are given below.

1) Type of FIS: Sugeno fuzzy model is used in this system. The Sugeno model is used because it is simpler in operation and it provides a continuous output control surface. The *first-order Sugeno* system is used. This means that the output is determined using a first order polynomial. The defuzzification method chosen is the *weighted average* method. This method is represented by the following formula:

$$z = \frac{\sum_i w_i z_i}{\sum_i w_i} \tag{1}$$

where w_i are the weights representing the strength of fired rules and z_i are the results of the first order polynomial function.

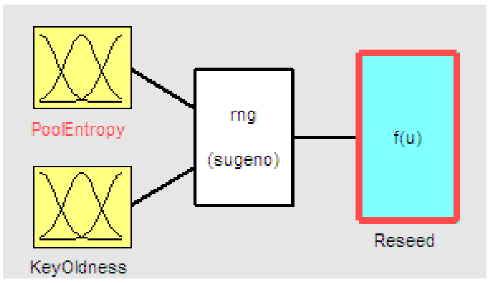


Fig. 2. Fuzzy Model

2) Inputs of FIS: The proposed fuzzy model takes two inputs, Pool Entropy ($data\_rate$) and Key Oldness ($key\_oldness$). The variable $data\_rate$ represents the rate at which data is currently being gathered from the random sources available. The variable $key\_oldness$ represents the number of times the 256-bit random output blocks have been generated by the generator core. There are three input membership functions (Low, Medium and High). These two inputs are continuously calculated by the generator core and passed onto the fuzzy inference system.

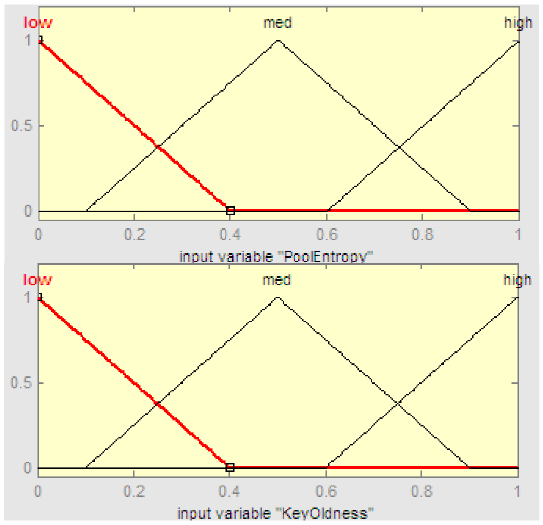


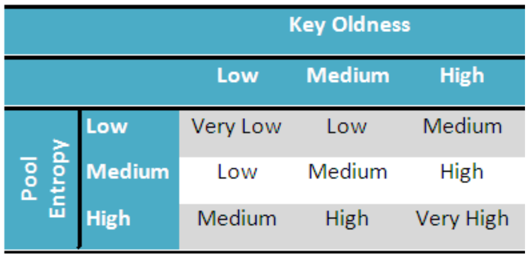
Fig. 3. Membership Functions for inputs

3) Output of FIS: The proposed fuzzy model gives a single output known as Reseed. When this output exceeds a threshold value, the reseeding process is done. There are five output membership functions (Very Low, Low, Medium, High and Very High). The membership functions of the output are defined by the following first order equations (where x is the input variable 'PoolEntropy', y is the input variable 'KeyOldness' and z is the output variable 'Reseed'):

TABLE I
EQUATIONS FOR OUTPUT 'RESEED'

Membership Function	*Sugeno Polynomial*
Very Low	$z_1 = 0.1x + 0.2y + 0.0$
Low	$z_2 = 0.2x + 0.2y + 0.2$
Medium	$z_3 = 0.3x + 0.3y + 0.4$
High	$z_4 = 0.2x + 0.2y + 0.4$
Very High	$z_5 = 0.1x + 0.1y + 0.8$

4) Rule base: The rule-base of the system consists of 9 rules. These rules have been built using intuition. The rules in the matrix form are shown below.



		Key Oldness		
		Low	Medium	High
Pool Entropy	Low	Very Low	Low	Medium
	Medium	Low	Medium	High
	High	Medium	High	Very High

Fig. 4. Rule Base

5) Surface Plot: The Surface plot of the fuzzy inference system is shown below. It shows that the fuzzy system is continuous and behaves as expected.

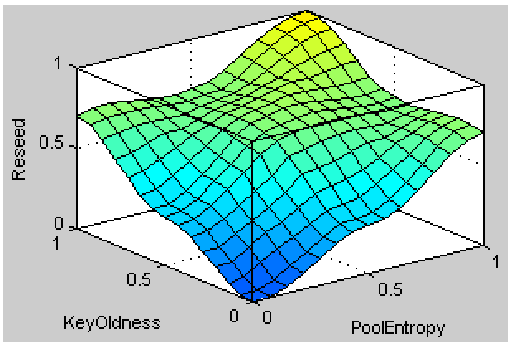


Fig. 5. Surface Plot

IV. Entropy Sources

Fuzz-Fortuna can accommodate up to 256 sources of entropy. In this implementation, we have used three entropy sources which are commonly available to all PC users.

The rate at which data is obtained from these sources is calculated by the following formula:

$$data\_rate_{new} = \alpha \times data\_rate_{previous} + (1 - \alpha) \times N_b \quad (2)$$

where N_b represents the number of bits obtained in the current run of the data gathering algorithm. The constant α is used to smooth the change in data rate. The typical value of α lies within the range 0.80-0.95 and is chosen on experimental basis.

The details of random sources used and analysis of the data obtained is given below.

A. Cursor Movement

Mouse is a commonly available device. It is a very low entropy source, and only the least significant bit can be considered unpredictable by the attacker[10]. The position of the cursor is defined in the Windows API as a structure of two long integers, x and y. The range of these values depend upon resolution of the screen. We used the resolution of 1024x768 on a 14 inch screen. We gathered and plotted different number of bits but the results for histogram and auto-correlation were best for the least significant bit of the x and y position as shown by the following figures. The data from cursor position is gathered only as long as it keeps moving. If the mouse stops for a certain amount of samples, it is not used as source until the cursor moves again.

B. Keystroke Timings

By sampling keystrokes every millisecond, we can get random data from the keystroke timings, ascii codes and scan codes of the keys pressed. However, only the two least significant bits of the keystroke timings are taken as random data, as they provide data with a uniform distribution.

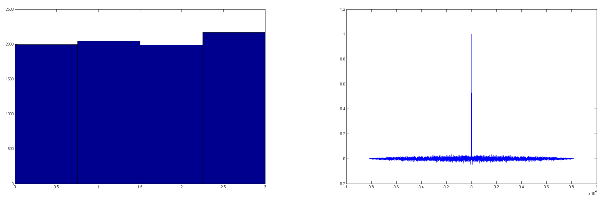


Fig. 6. Histogram and Auto-Correlation of Mouse Movement

C. Soundcard Noise

As mouse and keyboard require a user and are easily susceptible to an attack, the soundcard noise provides the complementary entropy source. In this implementation, a RealTek High Definition Audio soundcard was used. Eight-bit samples were taken at a sample rate of 800 samples per second. We gathered and plotted different number of bits but the results for histogram and auto-correlation were best for the two least significant bits of the obtained data. These results are shown in the following figures.

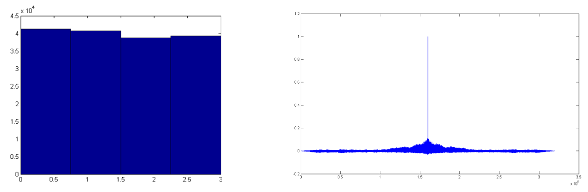


Fig. 7. Histogram and Auto-Correlation of Soundcard Noise

Data is gathered and concatenated after one second interval. The data full of entropy is then fed into the pools to increase their entropy.

V. The Fuzz-Fortuna Implementation and Algorithm

The Fuzz-Fortuna algorithm has been implemented in C++ language using MS Visual C++ 6.0 and MATLAB v.7.0. The pseudocode of the algorithm used in this implementation is given below.

VI. Results

The FuzzFortuna implementation resulted in a 696 KB executable. We were able to obtain a data rate of *8.42Mbps* on PC with processing power of 3.2 GHz and 1 GB RAM. To test the quality of random numbers generated by our implementation, we tested it using various methods which are described below:

A. Diehard Test Suite

Among the test-suites available on the World Wide Web for testing data for it randomness, Diehard test suite [11] is the most popular choice. Diehard test suite consists of 19 tests of randomness which can be performed on at least 12MB of data.

Algorithm VI.1: FuzzFortuna($seedfile$)

```
procedure InitializePRNG()
 global $Pools, Counter, PoolEntropy, KeyOldness$

procedure ReadSeed()
 $seed \leftarrow seedfile$
 return $(seed)$

procedure WriteSeed($seed$)
 $seedfile \leftarrow seed$

procedure Run_AES($value$)
 $encrypted \leftarrow$ AES_Encryption($value$)
 return $(encrypted)$

procedure AES_UpdateKey($key$)
 $AESkey \leftarrow key$

procedure AES_Counter($counter$)
 $RandomData \leftarrow$ Run_AES($counter$)
 $counter \leftarrow counter + 1$
 $KeyOldness \leftarrow KeyOldness + 1$
 return $(RandomData)$

procedure GetRandomSources()
 $Pools \leftarrow randomdatafromsources$
 $PoolEntropy \leftarrow size(datafromsources)$

procedure Reseed()
 if $PoolEntropy \geq size(seed)$
   then { AES_UpdateKey($seedFromPools$)
        { $PoolEntropy \leftarrow PoolEntropy - size(seed)$
        { $KeyOldness \leftarrow 0$
   else { AES_UpdateKey($seedFromOutput$)
        { $KeyOldness \leftarrow 0$

procedure Run_FIS($PoolEntropy, KeyOldness$)
 local $ThresholdValue$
 $Reseed \leftarrow$ FIS($PoolEntropy, KeyOldness$)
 if $Reseed \geq ThresholdValue$
   then Reseed()

main
 global $RandFile, RandFileSize, ReqSize$
 local $seed, randdatablock$
 InitializePRNG()
 $RandFileSize \leftarrow 0$
 $seed \leftarrow$ ReadSeed()
 Reseed($seed$)
 $randdatablock \leftarrow$ AES_Counter($counter$)
 WriteSeed($randdatablock$)
 while $RandFileSize < ReqSize$
   { GetRandomSources()
   { $randdatablock \leftarrow$ AES_Counter($counter$)
   { $RandomFile \leftarrow RandomFile \& randdatablock$
   { Run_FIS($PoolEntropy, KeyOldness$)
 return $(RandomFile)$
```

Diehard suits tests the statistical properties of generated data and compares them with the well established properties of true random data to give a measure of randomness of data. Each test returns P-values measuring the probability that a sample of the test data disagrees with the Null Hypothesis H_o that the test data has a particular distribution D. The P-values of zero or one for any test indicate that the tested data is not random. In order to compare Fuzz-Fortuna with the Fortuna algorithm implementation by [7], we use the scoring scheme by Johnson [12] which was also used by them. The scoring scheme as well as the results obtained are listed in the following tables.

TABLE II
DIEHARD SCORING SCHEME

P-Value	*Label*	*Score*
$0 < p < 0.95$	Good	0
$0.95 \leq p < 0.998$	Suspect	2
$p \geq 0.998$	Bad	4

TABLE III
SCORE COMPARISON FOR DIFFERENT RNGS

RNG	*Score*
'Mother'	20
True (TRNG)	22
FuzzFortuna	*20*
Fortuna	24
32-bit LFSR	162
EQG	288
LFSR	756
Maximum (worst) possible score	876

The improvement in the quality of random numbers generated by Fuzz-Fortuna over that of Fortuna indicates that the proposed algorithm is a better choice.

B. Entropy Calculation

To calculate the entropy content of the generated data, we used a program called 'ENT' [13]. ENT applies various tests to sequences of bytes stored in files and reports the results of those tests. The program is useful for evaluating pseudorandom number generators for encryption and statistical sampling applications, compression algorithms, and other applications where the information density of a file is of interest.

We got following results when ENT was applied to generated data in byte mode.
Entropy = 7.999996 bits per byte.
Optimum compression would reduce the size of this 41943040 byte file by 0 percent.
Chi square distribution for 41943040 samples is 233.12, and randomly would exceed this value 75.00 percent of the times.
Arithmetic mean value of data bytes is 127.5079 (127.5 = random).
Monte Carlo value for Pi is 3.141224970 (error 0.01 percent).
Serial correlation coefficient is 0.000091 (totally uncorrelated = 0.0).
These results prove the effectiveness and high quality of random data generated by our generator.

C. Compression Tests

When a data contains patterns, it can be compressed using data compression schemes. A truly random data should not be compressible. *To check for this property, we used a very popular compression software 'WinRAR' in its 'best' compression mode. WinRAR 'compressed' our 40 MB random data file in to 40.1 MB file which shows that the data file is completely uncompressible and the generated data is truly random.*

VII. CONCLUSION AND FUTURE WORK

This paper has presented the concept and implementation of fuzzified version of a cryptographically secure pseudorandom number generator namely Fuzz-Fortuna (originally known as Fortuna). The detailed analysis and results of tests done on the output of the developed system have been discussed. The results strengthen our assertion about the correctness and robustness of our proposed technique. The improvement in quality of generated random numbers certainly makes our proposed algorithm superior to the ordinary Fortuna algorithm.

In future, we look forward to adding more random entropy sources to the system, including those that are already being used for generation of random numbers by the */dev/random* device in the linux operating system. Fortuna is already being used as the preferred algorithm for */dev/random* device. We plan to present Fuzz-Fortuna as a replacement algorithm. The hardware implementation of Fuzz-Fortuna for use as random data source in custom designed chips is also an important goal.

REFERENCES

[1] N. Ferguson and B. Schneier, *Practical Cryptography*, Wiley Publishing Inc., 2003
[2] M. Mascagni, S. Cuccaro, D. Pryor and M. Robinson, *A Fast, High Quality, and Reproducible Parallel Lagged-Fibonacci Pseudorandom Number Generator*, Journal of computational physics, vol. 119, no. 2, pp. 211-219, Elsevier, 1995.
[3] L. Blum, M. Blum and M. Shub, *A Simple Unpredictable Pseudo-Random Number Generator*, SIAM Journal on Computing, vol. 15, pp 364, SIAM, 1986.
[4] Makoto Matsumoto and Takuji Nishimura, *Mersenne Twister: A 623 dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modelling and Computer Simulations, January 1998
[5] Mutsuo Saito and Makoto Matsumoto, *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*, MCQMC 2006
[6] John Kelsey, Bruce Schneier, and Niels Ferguson, *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic*, Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999
[7] Robert McEvoy, *Fortuna: Cryptographically Secure Pseudo-Random Number Generation In Software And Hardware*, ISSC 2006
[8] NIST, *Secure Hash Standard*, FIPS PUB 180-2, 2002
[9] NIST, *Advanced Encryption Standard*, FIPS PUB 197, 2001
[10] T. Matthews, *Suggestions for Random Number Generation in Software*, RSA Laboratories Bulletin #1. RSA Labs, Jan. 1996
[11] G. Marsaglia, *Diehard Battery of Tests of Randomness*, http://stat.fsu.edu/pub/diehard/
[12] B. C. Johnson, *Radix-b-extensions to some common empirical tests for pseudorandom number generators*, ACM Trans. Model. Comput. Simul., vol. 6, no. 4, pp. 261-273, 1996
[13] John "Random" Walker, *ENT: Entropy calculation and analysis of putative random sequences*, http://www.fourmilab.ch/random/