Two-Stage Random Generator (TSRG); Attack-Oriented Design and Implementation

Gamal Hussein, Yasser Dakroury, Bahaa Hassen, Ahmed Badr

Computer and Systems Engineering, Faculty of Eng.,
Ain Shams Uni., Cairo, Egypt
`Email:ams@idsc.net.eg`

Abstract

This paper presents the design principles and an implementation for one of the TSRG family members. The proposed new TSRG family utilizes the output of any PRNG called Randomizer as an auxiliary synchronized input to a second adapted suitable PRNG. TSRG output is modified to be a function of the output of the randomizer as well as its original variables. An instance of TSRG family, which consists of a cryptographic key stream randomizer (IDEA algorithm in OFB mode) and an adapted Lehmer generator, has been implemented as one of the possible family members. The implemented model components are studied with their rational and their working procedures are illustrated. An Attack-Oriented design considers the TSRG mathematical model and characteristics, the Randomizer systematic reseeding, the dynamic key updating, and the operating system features to inhibit all known PRNG attacks, enlarge TSRG overall period, and achieve functionally unpredictable PRNG output. To test its strength, all the TSRG defenses against attacks are thoroughly discussed and the major randomness tests are applied.

Keywords: PRNG, randomness, key generators, OTP, statistical tests, Lehmer, attack.

1. Introduction

It is hard to imagine a well designed cryptographic application that does not use random numbers generating secret values. For example, RNGs are required to generate session keys, initialization vectors, seeds, padding bytes and challenges. These values must be unknown to attackers. The one time pad, the only provably secure encryption system, uses as much key material as ciphertext and requires generated key stream from a truly random process [1]. Because security protocols rely on the unpredictability of the used keys, RNGs for cryptographic applications must meet rigorous requirements. Common PRNG design is based on heuristic approaches. These approaches consider PRNG as a program for generating a sequence of bits that passes a set of selected randomness tests. Since the set of randomness tests are arbitrary, it is possible to construct a set of efficient tests to foil the commonly used PRNG. Consequently, before using PRNG in a new application, extensive tests have to be conducted in order to detect the difference in behavior between the application with PRNG and the same application with True Random Data. In the case of using PRNG in cryptographic purposes, the situation becomes highly risky since the adversary may concentrate on the known weakness of the PRNG [2]. Random numbers are very difficult to generate, especially on computers, which are deterministic with low entropy input sources. This condition directs the design of PRNG towards distilling these sources to get high entropy values to be used in PRNG key parameters like the seed and keys. For example, Yarrow-160 [3] assumes it can accumulate enough entropy to get the PRNG in unguessable state. At this point, it is assumed that the cryptographic PRNG cannot be attacked while producing a limited number of output samples. Then, the PRNG must be reseeded to another unguessable state using new distilled entropy with a computationally expensive reseeding process to raise the cost of attempting to guess the PRNG's key. Non-deterministic output can be obtained through restarting the algorithm at unguessable instances with unguessable initial conditions and adequately large period. PRNGs are hardware, software or hybrid. Software generators are either based on using a mathematical formula relating the output of the generator with its previous outputs (e.g., congruential, BBS, MWC), or based on using encryption and/or hash techniques in a particular way to obtain unpredictable output (cryptographic PRNG). The first type suffers from relatively easy cracking as in the case of congruential equations [4] or relative slowness of generation time as in the case of BBS [5]. Cryptographic PRNGs depend on the strength of the used encryption technique and/or the one-way hash function as well as its design is predominantly ad hoc (i.e., not based on a mathematical model, without proof of security or randomness [6]). The concept of entropy is usually introduced to enhance PRNG against attacks, especially key compromise attacks. A set of good ideas and methodologies for designing PRNGs can be found in RFC 170. Bruce Schneier and John Kelsey also have introduced a very useful discussion of various cryptographic attacks on PRNG. Compounding and combining different PRNGs enhance their statistical output and enhance the resistance to cracking. To attack the compound generator, the determination of which output portion is related to which PRNG is the first step to crack, then the problem is reduced to cracking a single generator with known output. Fourier analysis may be very helpful in case of cracking combined generators [7, 8, 9].

Why a TSRG. True random numbers are needed for initial conditions and the whole sequence may depend on it as in the V1.1 Netscape browser flaw, which is based on seed compromise [10]. Consequently even these types of PRNG must have true random initial conditions.

PRNG attacks, specially key and internal state compromise, as well as the enormous resources of current processing power make the design of PRNGs very hard. Good PRNGs based on mathematical theories either suffer from easy cracking or long setup and/or generation time. Proprietary of some of strong PRNGs is another problem. Experience states that who makes security, can break it or even know shortcuts to do (e.g., trap door, S-boxes).

TSRG attack oriented design have been created using new available technology, especially Intel RNG, and appropriate mathematical models to produce functionally provably unpredictable and sufficient long period outputs. Its implementation model survives against attacks and suitable for the task it is made for (i.e., Pluggable Secured Email Client PSEC). At last, there is no such a thing as an ideal pseudo-random generator, but there are less and less excuses to use any generator based on an ad-hoc or obscure design.

Paper Outline. In Section 2, a definition of basic theories and the mathematical model on which TSRG is based are given. Section 3 discusses the statistical characteristics of the generator outputs and parameters affecting it and how it gives directions of the design approach. In Section 4, the components of the generator are described with their rationale. The evaluation of the generator defenses against all known attacks are illustrated in details in Section 5. Testing results are listed in Section 6 and finally the conclusion is presented in Section 7.

2. Two-Stage Random Number Generators

The basic idea of TSRG family is to suitably modify transition and/or output functions of a PRNG (adapted generator) to additionally depend on a new random variable (i.e., randomizer output). In cases of changing the PRNG output function by just combining its original output with the randomizer output using convenient computer operator, the family shrinks to combination generator. The implemented model adapts the Linear Congruential Generator (LCG) equation by replacing its constant term with the randomizer output. Dynamic reseeding of the cryptographic randomizer defends against attacks and makes the adapted generator works as PRNG period amplifier. At last, there is no such a thing as an ideal pseudo-random generator, but there are less and less excuses to use any generator based on an ad-hoc or obscure design [11].

2.1. TSRG Mathematical Model

Cryptosystems based on a One Time Pad (OTP) depend on identical copies of random data either generated from PRNGs or real pre-prepared true random data saved on their system database (usually on CDs). These copies are being securely exchanged between subscribers. The probability of copying these true random data and the ease of predicting the output of PRNGs, create the need to deduce PRNGs preserving the accepted statistical features, their outputs cannot be predicted and its generation time is accepted. The desired PRNG output must be reproducible with identical patterns at the parties. Its outputted OTP may be sent with the message, which means sending double the length of the message or sending a digest from which the original OTP can be reproduced. Suppose the following PRNG whose output is giving by $x_{n+1} = f(x_n, x_{n-1}, ...)$ and another PRNG with output $I(n)$, where n is the state number. x_{n+1} can be modified to: $x_{n+1} = f(x_n, x_{n-1}, ..., I(n+1))$. The output of this equation is not guaranteed to be random but it is worse to apply the concept to one of the well-known tested polynomial congruent equations (i.e., LCG). Lehmer proposed the congruential method of generating a sequence of pseudo-random numbers $x_1, x_2, x_3, ...$. In this method, each member of the sequence generates its successor by means of the following algorithm. The integer x_n is multiplied by a constant a then added to a constant c. The result is divided by a constant M, and the resulting remainder is taken as x_{n+1}. Thus

$$x_{n+1} \quad = \quad ax_n + c \qquad (\text{mod} M) \tag{1}$$

This parametric multiplicative LCG algorithm has withstood the test of time. It can be implemented efficiently, numerous empirical tests of the randomness of its output have been published, and its important theoretical properties have been analyzed [12]. Although Lehmer's algorithm has some statistical defects like ease of prediction of its parameters yet, if the algorithm parameters are chosen properly and the software implementation of the algorithm is correct, the resulting generator can produce a virtually infinite sequence of numbers that will satisfy almost any statistical test of randomness [13, 14]. The available theory is still somewhat incomplete and implicit. It has not emerged clearly how one calculates the period of sequence in the important case when the modulus M is a large prime or has a large prime as a factor (composite modulus) [15]. Newer empirical tests, especially in [16], show that some of other PRNGs like Combo, NCOMBO and Lagged-Fibonacci using *, are statistically better than LCG.

2.2. Basic Theorem

In the subsequent treatment, all numbers are integers, the modulus M is a prime number. Suppose a sequence of x's satisfies:

$$x_{n+1} \quad = \quad ax_n \qquad (\text{mod} M) \qquad (n = 0, 1, 2, ...) \tag{2}$$

The basic theory behind Lehmer's equation states that the sequence begins to repeat for a value of n which satisfies :

$$a^n \quad = \quad 1 \qquad (\text{mod} M) \tag{3}$$

Suppose the smallest positive integer n satisfying (3) is d. Then d is called the *period* of the sequence, and it must divide $(M-1)$ and be equal to it in case of a primitive root. All deductions concerning the multiplier primitive roots and composite modulus are thoroughly discussed in [15].

2.3. Modified Lehmer Equation

The objective of this section is to modify Lehmer's equation to survive attacks trying to predict its outputs using the previously read samples without sacrificing its statistical features. From equation (1),

$$x_n \quad = \quad a^n x_0 + a^{n-1}c + ... + c \qquad (\mathrm{mod} M) \tag{4}$$

Equation (4) has four unknowns (a, c, x_0, M), so four consecutive samples are enough to crack. In the proposed model, the randomizer is a key stream PRNG that consists of IDEA cipher algorithm in OFB mode. A relatively large true random seed which is called Basic Random Data (BRD) must be prepared. The BRD is fed in sequence to the encryption technique and its output is fed back again to the input of the encryption technique. The randomizer output is denoted by I. A set of operators and processes are defined as follows:

- BRD: Basic Random Data with length of 2048 bytes or 512 four-byte words.
- k_r: encryption key used in calculating the actual IDEA encryption key.
- H: Computationally expensive function of k_r and others, for simplicity denoted as $H(k_r)$ (see Section 4.3.4).
- IDEA: IDEA encryption algorithm operator with actual key $H(k_r)$. It operates on I_q at stage q to generate I_{q+1}. The length of the array I is 512 four-byte words.

Let $I_0 = BRD$, $||$ is the concatenation operator, $I = IDEA(BRD, H(k_r)) = IDEA(I_0, H(k_r))$, $I_2 = IDEA(I_1, H(k_r))$, in general:

$$I_{q+1} \quad = \quad IDEA(I_q, H(k_r)) \tag{5}$$

For any $q \geq 0$, I_q is 2048 bytes and is grouped in 512 four-byte words. The general equation for I is:

$$I_{q+1}(2i+1)||I_{q+1}(2i+2) \quad = \quad IDEA(I_q(2i+1)||I_q(2i+2), H(k_r)) \qquad (i = 0, 1, ..., 255) \tag{6}$$

TSRG output is generated by replacing the constant c in Equation (1) by the output of the randomizer I:

$$x_n \quad = \quad a^n x_0 + a^{n-1}I(1) + a^{n-2}(2) + ... + I_q(j) \qquad (\mathrm{mod} M) \tag{7}$$

where $n = 1, 2, ...$, $q = 1 + \lfloor (n-1)/512 \rfloor$, $j = n \bmod 512$ and $I_q(0) = I_q(512)$.

It is clear that Equation (7) is a generalized form of Equation (4) where the constant c is replaced by random values (the Is). The statistical features of samples generated by Equation (4) are accepted while c is constant. Since $I_q(j)$ is independent variable and based on random data then replacing cs with Is logically enhance or keep the PRNG's statistical features [17]. In fact, x_n is generated by combining (adding) delayed Lehmer PRNG outputs with random initial conditions. Marsaglia in [16] gives the theoretical support that the output of combining two or more simple generators, by means of a convenient computer operation such as $+$, $\times$ or exclusive-or, provides a composite with better randomness than either of the components. This explains why TSRG output in Equation (7) has better randomness features than LCG. To get the minimum number of samples to crack, the number of equations must equal the number of unknowns (Is, a, x_0, k_r, M). If the number of output samples is v, then:

$$\textbf{number of equations} = v + v/2 \quad = \quad \textbf{number of unknown variables} = 4 + v + 512 \tag{8}$$

The solution for v is 1032 and number of equations = number of variables = 1548 of which 516 are non-linear equations of the form (6). The previous set of equations cannot easily be solved depending on the strength of used encryption and hash algorithms. If the opponent has a solution for the previous number of equations, a limited number of bytes per message in the same session must be sent. Each x_n is a four-byte word. The opponent may guess, control or know part of BRD and k_r, use it in one of input attacks, so it is preferred to define a factor of safety $f > 1$ such that:

$$f = \frac{\textbf{Theoretical\_Required\_Samples\_to\_Crack}}{\textbf{Actual\_Required\_Ssamples\_to\_Crack}} \quad (9)$$

Implementation Notes.

1. Using IDEA encryption can be changed to other encryption technique. This would change the minimum number samples required to crack if the encrypted data word length differs from 8 bytes.

2. In this implementation,we have used Lehmer's equation but the concept can be applied to other PRNGs (e.g., congruential equations) to create random data of any length using limited size of basic random data.

3. The used Randomizer is a key stream PRNG which is slower one. It is appropriate to Email client application where speed is not the first issue. For application which need higher speeds, we may use other efficient Randomizer PRNG like Marsaglia, MWC or Lagged-Fibonacci PRNGs.

In all previous cases, all derivations about PRNG security and characteristics have to be redeveloped as will be shown in next sections.

3. TSRG Characteristics

In this section, the TSRG period, modular sum and arithmetic average over a complete period are summarized and compared to that of LCG. Attention is paid to the initial condition x_0 and its effect on the output samples.

3.1. TSRG Period

TSRG output is generated through Equation (1) by replacing the constant c with the output of the randomizer c_i. This implies that $x_{n+1} = ax_n + c_{n+1}$. For $r \geq 0$, the output sample x_r is given by:

$$x_r = a^r x_0 + a^{r-1}c_1 + a^{r-2}c_2 + ... + c_r$$

Let the randomizer output c_r be repeated after period s, so that $c_r + s = c_r$. Consequently

$$(a^{ms} - 1)\left(x_0 + \left(\sum_{i=1}^{s} a^{s-i}c_i\right)(a^s - 1)^{-1}\right) = 0 \qquad (\mathrm{mod} M) \quad (10)$$

It's easy now to deduce that the TSRG period p is given by Equation (11):

$$p = \textbf{Least\_Common\_Multiple}(\textbf{Randomizer\_period}, \textbf{Original\_Lehmer\_Generator\_Period}) \quad (11)$$

3.2. TSRG Modular Sum Over a Complete Period

To get the modular sum of a sequence, the sum of all output samples must be calculated over a complete period. The modular sum equals

$$-m\left(\sum_{I=1}^{s} c_i\right)(a-1)^{-1} \tag{12}$$

3.3. TSRG Average Over a Complete Period

By similar deductions, it can be proved that the average is sum/ms, where

$$sum \quad = \quad (a-1)^{-1}M\left[\begin{array}{l} (\lfloor a^{ms}/M \rfloor \left(ax_0+(a^s-1)^{-1}\left(\sum_{I=1}^{s} c_i a^{s+1-I}\right)\right) \\ -m\left(\sum_{I=1}^{s} c_i\right)(a-1)^{-1} \\ -M\left(\sum_{I=1}^{ms}\lfloor\frac{a^i x_0+\sum_{j=1}^{i} a^{i-j}c_j}{M}\rfloor\right) \end{array}\right] \tag{13}$$

It is clear that for the same randomizer and modified Lehmer generator, its biased average only depends on x_0.

3.4. TSRG Total Average Over a Complete Period

The total average is defined as the sum of all possible outputs over complete periods for all initial conditions divided by the number of summed elements. The sum of all initial conditions is the sum of all values from zero up to $M-1$, which equals $M(M-1)/2$. Figure 1 illustrates the process of the generation of TSRG for all initial conditions. All elements are modulo M.

x_0	1st sample	2nd sample	$\dots$
0	c_1	ac_1+c_2	$\dots$
1	$a+c_1$	$a^2+ac_1+c_2$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$
a^2x_0	$a^3x_0+c_1$	$a^4x_0+ac_1+c_2$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$
$M-1$	$a(M-1)+c_1$	$a^2(M-1)+ac_1+c_2$	$\dots$

Figure 1: TSRG Generation Process

The first column is clear to be all numbers from 0 to $M-1$, so its average is $(M-1)/2$. The second column consists of M elements of an LCG generator with constant c_1 and multiplier a and the sample $(1-a)^{-1}c_1$ $(\mathrm{mod} M)$. The average can also be proved to be $(M-1)/2$ and for large M the average converges to $M/2$. The same results can be proved for other columns. Here after a detailed example of a PRNG in the TSRG family whose randomizer is LCG with modulus 5 and multiplier 3 with initial condition 1 denoted as $L(5,3,1)$ and adapted Lehmer generator $L(7,5)$. Table 1 summarizes the results, which imply that 3 is the total average and 2 is the bad initial condition and TSRG period $p=(6\times 4)/2=12$.

Arth. Avr	x_0	Mod. Sum	O. Avr	O. Std	0	1	2	3	4	5	6
2.5833	0	3	1.714286	1.380131	2	4	1	0	1	3	1
2.5833	1	3	1.714286	1.380131	2	4	1	0	1	3	1
2	2	3	1.714286	1.380131	3	0	6	0	3	0	0
3.75	3	3	1.714286	1.380131	1	0	1	4	2	1	3
3.75	4	3	1.714286	1.380131	1	0	1	4	2	1	3
2.5833	5	3	1.714286	1.380131	2	4	1	0	1	3	1
3.75	6	3	1.714286	1.380131	1	0	1	4	2	1	3
3					12	12	12	12	12	12	12

Table 1: Two-Stage LCGs Statistics

Where: Arth. Avr = Arithmatic Average, O. Std. =Occurences Standard Deviation, Mod. Sum = Modular Sum, O. Avr = Occurences Average.

For the bad initial condition $x_0 = 2$, the period is reduced to 4 which is less than modulus 7, so not all set of outputs $0, ..., 6$ are generated. The output sequence has $X = 2$ with a rate double that of others (i.e., redundancy is higher). The apparent entropy of the output is highly affected, which may help in detecting bad initial conditions and trigger dynamic reseeding at shorter periods. The calculated total average is $(M - 1)/2 = 3$ which is unbiased. So, to obtain unbiased TSRG output, the randomizer is reseeded preserving the current x_n as new initial condition.

4. TSRG Design and Components

In this section, the design perspectives are introduced followed by TSRG implementation considerations, components and working procedure. The basic idea of TSRG is to use some initial random data (BRD; basic random data) and feed it to a block cipher such as IDEA to obtain random looking numbers c_i. Here, c_i is basically obtained by applying IDEA to c_{i-2048}. The key to IDEA is obtained by hashing previous key, internal state and TSRG outputs. The result is then used to feed a generalized Lehmer generator to generate output. The entropy estimators on the x_i and c_i are used to control the period of a specific key before its updating. A systematic randomizer reseeding occurs every M samples to unbias TSRG output and enlarge its period.

In the next subsections, the components of the proposed generator are discussed in detail.

4.1. Design Perspectives

A PRNG depends on cryptographic or polynomial algorithms with random initial conditions or continuous distilled sources of entropy. In the last case, RNG can be designed using collecting and distillation of enough entropy from various independent sources. If the number of independent sources are very large and the entropy distillation of them is accurate then a nearly perfect RNG is obtained. If the sources have no new entropy bits, a choice is available to switch to cryptographic mode until enough entropy is available or stop producing output. Stopping producing output is perfect but not practical. Cryptographic mode associates the ability to be attacked depending on the number of samples cryptographically produced and the used generation technique. In the OTP cryptosystem, identical copies of the random data must be available to the sender and receiver. If PRNG is used for OTP cryptosystem then identical objects including data and programs must be available to them both. In case of using continuous sources of entropy, the receiver must know exactly the points of switching and the new states after switching to cryptographic mode. This can be done using a special script language. The format of the script language may consist of delimited entropy bits concatenated with number of samples to be produced using new resulting state. The length of script program is less than the length of the outputted generated random data and the ratio between them is inversely proportional to number of switches. The minimum value of the previous ratio is obtained when PRNG permanently work in cryptographic mode which can be attacked. In this case the script program is one record that consists of the initial condition and the number of the desired PRNG output samples. If the switching is very intensive then the script program length may be huge and comparable with the length of random data length itself.

The very pretty Yarrow PRNG [3] suffers from this problem when it is used in OTP security as well as its ad hoc design. This means there must be another approach with minimum sacrifices. The proposed approach is to gather enough entropy and save it in what is called a limited length Pool and use it adequately to generate cryptographic random data while dynamic reseedings to unguessable states are occurred before generating enough samples sufficient to crack. The Pool of random bits (BRD and IDEA key) is securely transferred to receiver. The number of samples sufficient to crack is achieved from the generator mathematical model and dynamic reseeding to unguessable state is made using computationally expensive calculation depending of the apparent entropy of previous internal states and output of the generator itself. TSRG has two types of reseeding mechanisms. The first changes the key of the randomizer to a new unguessable state. The other occurs after every M samples to force the cycle length of the randomizer to M. In this case the initial pool is reloaded while the current x_n is used as initial condition to the next run.

4.2. TSRG Implementation Considerations

The coding of PRNGs is another source of weaknesses. The programmer must be sure that the output of the generator is exactly the same as he expects. This can be done using well tested composing modules and testing vectors. TSRG composing modules are LCG module, IDEA cipher technique, Intel RNG driver [18] and FIPS-140-1 randomness testing modules. All the previous modules are produced by reliable sources and published in source code on the Internet to be reviewed and tested [16, 19, 20, 21]. Marsaglia diehard rigorous randomness testing modules includes source code for all well known polynomial generators and the crypto32 library contains all known ciphering techniques including IDEA. The Intel driver can be downloaded free from Intel's site. In designing TSRG, the following factors are guidelines that are considered [22]:

1. Attack-Oriented design: which means that every known attack is considered while designing process.
2. Use of well tested primitives (encryption, LCG, Intel RNG driver,...).
3. Base the PRNG on something strong (IDEA is strong encryption technique and not old like DES and not new like AES).
4. Independent components: which can be easily maintained and /or modified.
5. Efficiency : each needed component is implemented however its performance degradation.
6. Resist backtracking: Key updating through irreversible process.
7. Resist Chosen-Input Attacks : PRNG inputs is scattered in the application (PSEC) to resist control and then internally tested for randomness.
8. Quick Recover from Compromises: key lasts for controlled period that is accepted by the designer to be useless for the attacker.
9. Dynamically generate a new starting PRNG state: The used hash functions help in switching to unguessable sate with dynamic periods that depend on entropy of selected points.
10. Good statistical characteristics (large period, pass randomness tests, ...).
11. Theoretical Support: It is not ad hoc design. Characteristics (like repeatability, portability, jumping ahead, ease of implementation, easy to generate, hardness of prediction and memory and speed requirements) are thoroughly discussed and evaluated.

4.3. TSRG Components

The TSRG components are shown in Figure 2. It basically consists of Randomizer, Modified Lehmer Generator, Entropy Estimator, Key Updating module and Reseeding Mechanism.

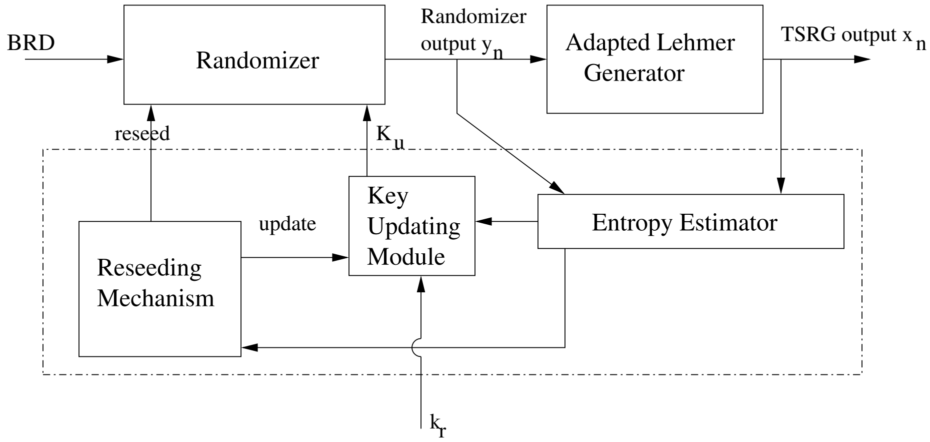

Figure 2: TSRG Block Diagram

4.3.1. Randomizer

The randomizer is stream generator designed using encryption technique in feedback mode. Its initial vector is not one encryption block (8 bytes in case of IDEA cipher) but it is 2048 bytes that are fed to be encrypted in sequence then fed back to the input again. The output of encryption algorithm in OFB mode after every iteration can be approximated by a random-draw-without-replacement process. From the previous correspondence the average cycle size for DES encryption in OFB mode without key reseeding nearly equals 2^{63} [23, 24]. While the previous result is made by approximation and for DES, it is usually stated in general for other encryption techniques that "the average cycle size of encryption technique with input block of size m bits in OFB mode equals 2^{m-1}". The theoretical and actual experiments performed by Davis and Parkin proved that OFB with full output block is more secure than using OFB of part of the output [24]. In our model, the BRD array is divided into input blocks which are fed to the encryption technique in sequence and its corresponding output is fed again at its order in its turn. It is clear that the average cycle size in this case is not as that of OFB of one block. This issue is under consideration and will be dealt in further study. The use of multi-block here is made to make the generator resist key-compromise attacks and even the case when the attacker has a tool to break the encryption algorithm itself. Referring to the mathematical model in the previous section, the set of output samples using specific key before reseeding is not sufficient to solve the set of equations independent of the encryption technique used or whether it can be cracked or not.

Implementation Notes. While randomness testing of many security techniques is well tested by NIST, especially the finalist candidates (Twofish, Serpent, Rijndael, RC6 and Mars) [25], IDEA is the implemented one in TSRG.

IDEA is the encryption technique used to in OFB mode for the following reasons:

1. The TSRG implementation allows any other encryption technique to be easily implemented.

2. It is designed to be efficient in software, encrypting 64 bits with 128 bits key [26].

3. It is well tested with sufficiently published results since development 1990.
4. Its weak keys are studied and published as in [27]. The key verification module is not implemented yet.

4.3.2. Modified Lehmer Generator

The task of this module is to ensure randomness of the output and add complexity for cracking trials. Another advantage is that it works as period amplifier with amplification factor M. The paid processing cost can be relatively neglected relative to encryption processing time. Since the randomizer is reseeded every M samples and using Equation (11) the period of TSRG is in the order of M^2. After reseeding, the Lehmer initial condition is the last generated one to amplify the period by M times. For all initial conditions with fixed key updating and randomizer reseeding every M samples, the expected period in the order of M^3 and the output is unbiased.

4.3.3. Entropy Estimator Module

The task of this module is to calculate the apparent entropies of the randomizer and TSRG outputs. Key updating module and reseeding mechanism exploit its outputs using Equations (15, 14) respectively. TSRG and Randomizer output apparent entropies values can be calculated using the equation $H = -\sum(p \log p)$, where p is probability of generation of any character. And the function $\sum$ is applied for all characters from ASCII 0 up to 255. The apparent entropy calculations are done as the output is generated in the following manner:

1. Two 256 entry tables (entropy tables) are allocated to randomizer and TSRG outputs respectively and all their entries are initialized to zero.
2. Each character is generated either in randomizer or TSRG, its ASCII value is calculated and its entry in its table whose address equal to that ASCII is incremented by one.
3. At specific points, when the entropy value is required to be calculated ,the probability p of each character is calculated which equals to the value of its entry divided by the sum of all entries of its table.
4. Calculate the entropy $H = \sum(-p \log p)$ for 256 characters.

The specific points of calculating the entropy is one of the adjustable system parameters, In TSRG implementation it is adjusted to test at half of expected key update period defined in Equation (14).

At TSRG starting, the two entropy tables are initialized to zero (reseeded) and BRD entropy is calculated. The randomizer entropy and TSRG entropy are assigned to the BRD entropy.

The first estimate of key reseed period is calculated. After one half of it another trial, based on the generated output, is performed. When the key update period is encountered, the key is updated to unguessable state and the entropy tables are reinitialized.

4.3.4. Key Updating Module

It is responsible for changing the key k_r to an unguessable state key k_u using computationally expensive functions. Updating to new state uses TSRG output combined with randomizer output (not observable by the attacker) and current key to obtain new key. Although key updating occurs every period pk determined by Equation (14), it is prepared while the output is generated.

$$
\begin{aligned}
pk &= \textbf{Key\_Update\_Period} \qquad (14)\\
&= \left(\frac{\textbf{Min\_Samples\_To\_Crack}}{\textbf{Factor\_Of\_Safety}}\right) \times \textbf{Randomizer\_Output\_Apparent\_Entropy}\\
&\qquad \times \textbf{TSRG\_Output\_Apparent\_Entropy}
\end{aligned}
$$

The first two factors are given in Section 2. In Equation (14), we assume linear dependency between key updating periods and apparent entropies (less than 1). Bad initial conditions and randomizer short cycles force the key updating at smaller periods. The new key is produced by Equation (15):

$$k_u(z+1) \quad = \quad hash(k_u(z)||(x_0||y_0)||(x_1||y_1)||...(x_{pk}y_{pk})) \tag{15}$$

Where $hash$ is a modified SHA hash function returning 128 bits, z is the update number, x_{pk} and y_{pk} are last output samples before key updating as defined in Equation (14).

4.3.5. Reseeding Mechanism

This module has two functions, the first is to reseed the randomizer every M samples and the second to dynamically trigger key updating. Randomizer reseeding uses the value of the output x_n as the new initial condition of the next run to get unbiased output.

4.4. TSRG Working Procedures

The following Algorithm describes the generation mechanism for TSRG.

```
Algorithm TSRG(Tsrg-Length,X0)
  Generator_Initial_Condition = X0
  Re_Seed ()
  Tsrg_Absolute_Counter = 0
  Ku_Temp = Kr
  DO WHILE Tsrg_Absolute_Counter <= Tsrg_Length
    Yn = Generate_Randomizer_Sample ()
    Xn = Generate_Generator_Sample ()
    Ku_Temp = hash (Ku_Temp || Xn||Yn)
    Modular_Increment (Tsrg_Moduler_Counter)
    IF Toggle
      Update_Randomizer_Table()
      Update_Generator_Table()
      IF (Tsrg_Modulr_Counter >= 0.5*Key_Update_Period)
        Toggle = False
        Recalculate_Key_Update_Period()
      ENDIF
    ENDIF
    IF MOD(Tsrg_Absolute_Counter = 0)
      ReSeed()
    ENDIF
    IF (Tsrg_Modulr_Counter >= Key_Update_Period)
      Update_Key ()
    ENDIF
  ENDDO
END TSRG

Algorithm Re_Seed ()
  Toggle = True
  Randomizer_Input_Data = Basic_Random_Data
  Er = Apparent_Entropy (Randomizer_Input_Data)
```

```
  Eg = Er
  Key_Update_Period = Max_Samples*Eg*Er/Factor_Of_Safety
  Tsrg_Module_Counter = 0
  Initialize (Generator_Entropy_Table)
  Initialize (Randomizer_Entropy_Table)
END Re_Seed

Algorithm Recalculate_Key_Update_Period ()
  Er = Apparent_Entropy (Generator_Entropy_Table)
  Eg = Apparent_Entropy (Randomizer_Entropy_Table)
  Key_Update_Period= Min_Samples_To_Crack*Eg*Er/Factor_Of_Safety
END Recalculate_Key_Update_Period

Algorithm Update_Key ()
  Toggle = True
  Randomizer_Key= Size_Adaptor (Ku_Temp)
  Er = Apparent_Entropy (Randomizer_Input_Data)
  Eg = Er
  Key_Update_Period= Max_Samples*Eg*Er/Factor_Of_Safety
  Tsrg_Module_Counter = 0
  Initialize (Generator_Entropy_Table)
  Initialize (Randomizer_Entropy_Table)
END Update_Key
```

5. TSRG Attack Defences

The most important issue is that attackers, including those who know the RNG design, must not be able to make any useful predictions about the RNG outputs before and after the samples he already knows. The apparent entropy of the RNG output should be as close as possible to the bit length itself. The previous statement is essential but not sufficient since many statistically accepted PRNGs are easily cracked. For example, techniques are now known for breaking all polynomial congruent generators [3, 4, 7, 8]. Since there is no universally applicable and practical criteria to certify the security of PRNGs, attack defense against all known attacks may be employed for this task that holds true till new attacks appear or some security factor is discovered to be overestimated. In the following, a discussion of how the proposed generator resists various attacks:

5.1. Direct Cryptanalysis Attack Defence

Most cryptographic PRNGs mainly depend on the strength of the encryption and hash techniques to defend against this attack (e.g., ANSI9.17, DSA). TSRG outputs as shown in Section 2, cannot be used to mount this attack since there are enough read samples to solve the set of equations irrespective of the used encryption and hash functions. The expensive key updating algorithm forces the attacker to solve a new set of equations with number of unknowns greater than the number of equations. This means that TSRG does not depend entirely on the encryption and hash algorithms to defend against this attack.

5.2. Input Attack Defence

TSRG gets its input from three sources (Intel RNG, mouse movements and key strokes timings). It accepts 2500-byte input only once from Intel RNG. Maximized number of apparent random bits, gathered from mouse movements and key stroke timings then distilled. Intel RNG high entropy source data is combined with distilled data of the low entropy sources. Although Intel RNG output is guaranteed to be accepted using FIPS 140-1 randomness test, the combined data must be tested again, using FIPS 140-1, in this module to be sure that no attacker demon program is mounted to emulate generation of the required random data. An exclusive-or between random data from the Intel generator and that of low entropy sources is used to combine them. The exclusive-or is not securely perfect but its response time is optimum. The tested combined data is used to generate BRD and k_r. Testing the randomness of BRD and scattering entropy gathering modules in the application defends against chosen-input attacks. If the attacker knows all three sources, there is no cryptanalysis to be performed. If he knows part of them and has a direct observation of TSRG output, he has a chance to mount combined cryptanalysis known-input attacks.

5.3. Combined Cryptanalysis Known-Input Attacks

Suppose the attacker knows part of the inputs which can be a portion of BRD and at the same time TSRG output can be read. Knowing the TSRG design, if the portion of BRD greater than $1/f$ defined in Equation (14) the attack can succeed. This explains why we introduce this factor of safety parameter in this equation.

5.4. State-Compromise Attack Defence

Suppose that, for whatever reason, the adversary manages to learn the internal state I, exploiting some inadvertent leak, temporary penetration, cryptanalysis success, etc. Once the state is known, one or more of the following attacks can be mounted.

5.4.1. Backtracking Attacks

A Backtracking attack uses the compromise of the PRNG state I at time t to learn previous PRNG outputs. Key updating algorithm is an irreversible process through using truncation of the output of the hash function. So, it is impossible to get the previous key, hence the previous samples, when state-compromise attack has succeeded. This is another reason for using IDEA encryption technique whose key is 128 bits while the output of the hash function is 160 bits.

5.4.2. Permanent-Compromise Attacks

A permanent compromise attack occurs if, once an attacker compromises I at time t, all future and past I values are vulnerable to attack. Past values cannot be predicted as shown in Section 5.4.1 but future values can succeed if TSRG output can be observed or calculated (e.g., the parameters M and a in Lehmer's equation are known). TSRG does not process input and so can never recover from state compromise attack. It is clear that TSRG should resist state compromise extension attack as thoroughly as possible. In the proposed implementation model, the internal state and key variables are saved in Windows virtual memory. Memory manager swaps them from memory page to the hard disk of which the state can be compromised. State compromise attack may be resisted by allocating the internal state variables (Key and Is) in memory pages that are extensively referenced. The following code shows the memory requirements for TSRG static data segment and how the attack can be avoided.

```
BYTE Generator_Entropy_Table [256];
BYTE Basic_Random_Data [2048];
BYTE Key_Random [16];
BYTE Key_Updated [16];
BYTE Internal_State[2048];
BYTE Randomizer_Entropy_Table[256];
```

For every generated character, the entropy tables are updated avoiding probable disk swapping of this page.

5.4.3. Other State-Compromise Attacks

Iterative-guessing attacks and meet–in–the–middle attacks usually associated with input attacks are not applicatable to TSRG.

6. TSRG Randomness Testing

In this part, four randomness testing programs (OTP [28], ENT [20], FIPS140-1,2 [29, 30] and Diehard [31]) are used to evaluate randomness of TSRG output. With the same BRD, initial condition and initial key, different lengths of outputs are generated to observe changes of randomness test results. The 2500-byte array from which all initial conditions are obtained must pass FIPS140-1 test. The same test is applied to every TSRG 2500-byte output blocks. Applying ENT program to 40000-byte sample output result in passing the test. 1 mega bytes random output passes FIPS 140-2 tests by using OTP program. For Marsaglia Diehard test, the following table 2 summarizes the result compared to original LCG. The result of the above tests is "TSRG passes all randomness tests while others may fail".

	LATTIC	PARKLOT	MTUPLE	OPSO	BDAY	OPERM	RUNS	RANK
LCG	Fail	PASS	Fial	Fail	PASS	PASS	PASS	PASS
TSRG	PASS	PASS	PASS	PASS	PASS	PASS	PASS	PASS

Table 2: Diehard Randomness Test Results

7. Conclusion

TSRG is a proposed approach for the chaining of two interacting PRNGs. The first stage generator is any PRNG (Randomizer) whose output is fed to modify transition and/or output function of the second stage PRNG (adapted generator) to additionally depend on randomizer output. Many of the well known families of PRNGs (i.e., shrinking, compound and combination PRNGs) can be considered as instances of the TSRG family. TSRG characteristics (repeatability, jumping ahead, ease of implementation, ease of generating, hardness of prediction, memory and speed requirements) are studied and evaluated. One of the implemented family members, a hybrid PRNG, is obtained through modifying a Lehmer generator with the output of a cryptographic randomizer. The implemented randomizer is a key stream generator that uses IDEA algorithm in OFB mode with dynamic key updating and synchronous reseeding to inhibit output prediction, get unbiased output and elongate the TSRG overall period. The attack oriented design of this instance is thoroughly considered and studied. Cryptanalysis attack is defended by reseeding the randomizer before the minimum number of samples sufficient to crack, based on the deduced mathematical model. Dynamic Key updating resists permanent-compromise, backtracking and iterative-guessing attacks. Testing the randomness of the BRD fed as input to the randomizer from the Intel RNG generaor and scattering the entropy gathering modules in the implementation defends against chosen-input attack. Depending on the application environment (e.g., PRNG-based cryptosystem), the required characteristics can be obtained from a suitable selection of the randomizer and a clever modification of the adapted generator function.

References

[1] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, vol. 28, October 1949, pages 656-715. `http://www3.edgenet.net/dcowley/docs.html`.

[2] O. Goldreich. *Foundations of Cryptography*. Weizman Institute of Science and Applied Mathematics, Rehovot, Israel, 2001. `http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html`.

[3] J. Kelsey. B. Schneier and N. Ferguson. Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. *Sixth Annual Workshop on Selected Areas in Cryptography*, Springer Verlag, August 1999. `http://www.counterpane.com/yarrow-notes.html`.

[4] H. Krawczyk. How to predict congruential generators. *Journal of Algorithms*, vol. 13, no.4, December 1992.

[5] T. Ritter. Random/pseudo random number generator. `http://www.ciphersbyritter.com/NEWS2/94061801.HTM`. `sci.crypt` forum, June 1994.

[6] J. Håstad and M. Näslund. Key feedback mode: A keystream generator with provable security. NIST, 2000. `http://csrc.nist.gov/encryption/modes/proposedmodes/kfb/kfb-spec.pdf`.

[7] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. RFC 1750, Network Working Group, December 1994. `http://www.ietf.org/rfc/rfc1750.txt`.

[8] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption, Fifth International Workshop*. Springer Verlag Lecture Notes in Computer Science 1372, March 1998, pages 168–188. `http://www.counterpane.com/pseudorandom_number.pdf`.

[9] R.R. Coveyou and R.D. Macpherson. Fourier analysis of uniform random number generators. *Journal of the ACM*, vol. 14, no. 1, January 1967, pages 100–119. `http://portal.acm.org/citation.cfm?id=321379&coll=portal&dl=ACM&CFID=1840718&CFTOKEN=33814197`

[10] D.A. Pinnavaia. Generating good random numbers. `http://www.rit.edu/~dap0637/numbers.html`. Cryptography 0603-482, Fall 2001.

[11] T. Moreau. Pseudo-random generators, a high-level survey-in-progress. CONNOTECH Experts-conseils Inc., March 1997. `http://www.connotech.com/rng.htm`.

[12] C.H. Sauer and K.M. Chandy. *Computer Systems Performance Modeling*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[13] W. Stallings. *Network and Internetwork Security*. Prentice-Hall, Englewood Cliffs, N.J., 1995.

[14] S.K. Park and K.W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, vol. 31, no. 2, October 1988.

[15] A.T. Fuller. The period of pseudorandom numbers generated by Lehmer's congruential method. *The Computer Journal*, vol. 19, no. 2, 1976.

[16] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics: Proceedings of the 16th Symposium on the Interface*, Atlanta, Georgia, USA, March 1984. L. Billard, ed. Elsevier, New York, 1985. `http://stat.fsu.edu/~geo/diehard.html`.

[17] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill Book Company, 1965.

[18] B. Jun and P. Kocher. The Intel random number generator. Cryptography Research Inc., April 1999. `http://developer.intel.com/design/security/rng/rng.htm`.

[19] W. Dai. Crypto++, version 4.2. `http://www.eskimo.com/~weidai/cryptlib.html`.

[20] J. Walker. A pseudorandom number sequence test program. October 1998. `http://www.fourmilab.ch/random/`.

[21] Intel Inc. Intel 820 chipset system driver and utilities. http://www.intel.com/design/software/drivers/platform/820.htm.

[22] P. Gutmann. Software generation of practically strong random numbers. In *7th USENIX Security Symposium*, San Antonio, Texas, USA, 1998. http://www.usenix.org/publications/library/proceedings/sec98/gutmann.html.

[23] R.R. Jueneman. Analysis of certain aspects of output feedback mode. In D. Chaum and R.L. Rivest and A.T. Sherman, eds., *Advances in Cryptology: Proceedings of Crypto'82*, pages 99-127, August 1982. Plenum Press, New York and London, 1983. http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/C82/99.PDF.

[24] D.W. Davies and G.I. Parkin. The average cycle size of the key stream in output feedback encipherment. In D. Chaum and R.L. Rivest and A.T. Sherman, eds., *Advances in Cryptology: Proceedings of Crypto'82*, pages 99-127, August 1982. Plenum Press, New York and London, 1983. http://cnscenter.future.co.kr/resource/crypto/algorithm/block/cr82Davies.pdf.

[25] J. Soto. Randomness testing for the Advanced Encryption Standard finalist candidates. March 2000. http://csrc.nist.gov/rng/aes-report-final.doc.

[26] C. Kaufman, R. Perlman, and M. Speciner. *Network Security, Private Communication in a Public World*. Prentice-Hall, 1995.

[27] J. Daemen. *Cipher and Hash Function Design, Strategies Based on Linear and Differential Cryptanalysis*. Doctoral Dissertation, Katolische Universiteit Leuven, Belgium, March 1995. http://www.esat.kuleuven.ac.be/~cosicart/ps/JD-9500/.

[28] One Time Pad Encryption —0.9.4— Test Run. http://www.vidwest.com/otp/testrun.htm.

[29] N. Bridges. FIPS140-1, FIPS140-2 randomness tests. http://www.quartus.net/files/Misc/fips140x.zip.

[30] W. N. Havener, R. J. Medlock, L.D. Mitchell, and R.J. Walcott. Derived test requirements for FIPS PUB 140-1, security requirements for cryptographic modules. Part 1. NIST, March 1995. http://csrc.ncsl.nist.gov/cryptval/140-1/140test1.htm.

[31] G. Marsaglia. Diehard battery of tests. http://wwws.irb.hr/~stipy/random/Diehard.html.