Hindawi
Complexity
Volume 2017, Article ID 7436709, 17 pages
https://doi.org/10.1155/2017/7436709

WILEY | Hindawi

Research Article

An Improved Genetic Algorithm for Developing Deterministic OTP Key Generator

Ashish Jain[1] and Narendra S. Chaudhari[1,2]

[1]*Discipline of Computer Science and Engineering, Indian Institute of Technology Indore, Indore, India*
[2]*Discipline of Computer Science and Engineering, Visvesvaraya National Institute of Technology Nagpur, Nagpur, India*

Correspondence should be addressed to Ashish Jain; ashishjn.research@gmail.com

Received 22 March 2017; Revised 28 June 2017; Accepted 10 July 2017; Published 11 October 2017

Academic Editor: Roberto Natella

Copyright © 2017 Ashish Jain and Narendra S. Chaudhari. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Recently, a genetic-based random key generator (GRKG) for the one-time pad (OTP) cryptosystem has been proposed in the literature which has certain limitations. In this paper, two main characteristics (speed and randomness) of the GRKG method are significantly improved by presenting the IGRKG method (improved genetic-based random key generator method). The proposed IGRKG method generates an initial pad by using linear congruential generator (LCG) and improves the randomness of the initial pad using genetic algorithm. There are three reasons behind the use of LCG: it is easy to implement, it can run efficiently on computer hardware, and it has good statistical properties. The experimental results show the superiority of the IGRKG over GRKG in terms of speed and randomness. Hereby we would like to mention that no prior experimental work has been presented in the literature which is directly related to the OTP key generation using evolutionary algorithms. Therefore, this work can be considered as a guideline for future research.

1. Introduction

Recent years have witnessed use of information in many areas including financial accounts, military and political. Security of this information in both storage and transit is crucial as it may be compromised resulting in financial loss, disclosure of military or commercial secrets, and even the loss of life. Cryptography is one set of techniques for providing information security. Historically, cryptography is commonly connected with surveillance, warfare, and the similar applications. However, with the advent of information civilization and digital revolution, cryptography is also useful in the peaceful lives of common people, for example, when buying something over the Internet through credit card, withdrawing money from the ATM machines using smartcards, and locking and unlocking luxury cars.

Cryptography is related to the design of cryptosystems. Cryptosystems have two divisions: symmetric key and asymmetric key. In the case of symmetric-key cryptosystem, encryption function takes a text message (plaintext) as input and transforms it into an unreadable text (ciphertext) with the use of a secret key [1]. The decryption function converts the ciphertext back to the plaintext using the same secret key. If any flaws or oversights exist in the cryptosystem, it can be exploited by the attacker [1]. The attacker can recover the plaintext from the ciphertext without knowing the secret key because of openness of cryptographic algorithms and the encrypted data transfers via the insecure public communication channel. For this reason, sensitive applications, for example, financial domain, demand perfect security that can only be achieved by one-time pad (OTP) symmetric-key cryptosystems in which the key used for encryption once is never used anymore at any time [1]. For achieving perfect security, an obvious choice is random generation of the key via truly random sources. However, this choice is inefficient (generation of truly random numbers from hardware-based physical phenomena, for example, elapsed time between emissions of particles during radioactive decay; thermal noise from a semiconductor diode or resistors; sound from a microphone or video input from a camera, and so on; and generation of truly random numbers from software-based process, for example, the system clock; elapsed time between

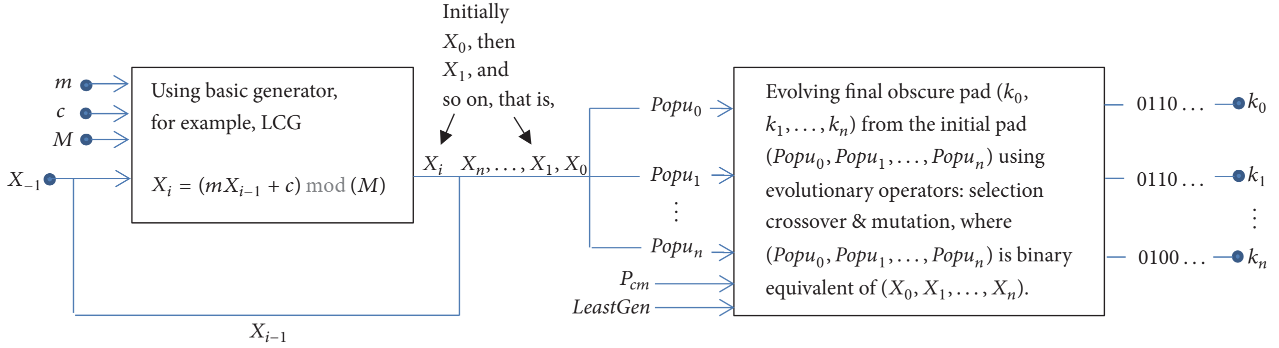


Figure 1: Block diagram of the proposed generator: IGRKG.

keystrokes or mouse movement; and operating system values such as system load and network statistics are impractical choices for practical cryptographic applications, i.e., large sized keys for each encryption [1]). Therefore, for sensitive applications pseudorandom generation of the key is the only option to make the scheme practical. Recent years have witnessed large use of computationally secure OTP all over the world, typically during financial transactions. Hereinafter OTP key means computationally secure pseudorandom key and original OTP key means truly random key. In this work, we present a genetic-based scheme for automatically generating OTP keys.

2. Related Work and Our Contributions

Several things in the world are naturally encoded, for example, genomes of animals [2]. This motivates us to utilize *(genotype)* genetic algorithms in development of deterministic scheme that can generate the OTP keys rapidly. In 2013, Sokouti et al. [3] have demonstrated a significant use of genetic algorithm for automatically generating OTP keys. They have proposed and compared two genetic-based OTP key generators, namely, 10P-GRKG and the GRKG. The comparison results in [3] show that the GRKG method is much better than the 10P-GRKG method in terms of speed and randomness. However, it is observed that the GRKG method has certain limitations which needs improvements. In this paper, we propose an improved genetic-based random key generator (IGRKG). As compared to GRKG, the proposed IGRKG generator generates the OTP key rapidly and the degree of randomness of the generated keys is better. In the literature, a prior attempt in OTP key generation using evolutionary technique has been addressed only in [3]. Therefore, this paper can present a detailed comparison between GRKG and IGRKG generators. We also compare the Diehard scores of GRKG and IGRKG with some existing pseudorandom number generators. It is important to note that, except GRKG and IGRKG generators, the other pseudorandom generators have not been developed to generate OTP key.

It should be noted that speed and randomness are the main objectives of a designer behind the design of a pseudorandom key generator. For achieving these objectives the following novelties and modifications are introduced which are our major contributions:

(1) Unlike GRKG we use a comparatively short secret key.

(2) Unlike *MaxGen* parameter employed in GRKG, a new *LeastGen* variable is proposed, where the essence of the parameter is the minimum generations in which the initial pad is obscured almost entirely.

(3) For determination of the crossover point, rather than using modular arithmetic over addition, we introduce a new approach of modular arithmetic over subtraction. The advantage of this approach is that it improves the randomness of the pad and makes the scheme faster.

(4) For evolving existing generation, a new and efficient approach is introduced that updates two variables π_1 and π_2. These variables are employed in Algorithm 1, Steps (6)–(14), to decide crossover and mutation points. This idea increases the randomness of the existing pad and evolves the obscure final pad rapidly.

(5) For increasing the speed of encryption and decryption, a more efficient encryption and decryption function is suggested.

Figure 1 shows the block diagram of the proposed work. Figure 1 shows that four integer values are taken as input corresponding to the short secret key: X_{-1}, c, m, and M. It should be noted that the values of these parameters are taken only *once* in the presence of both the sender and receiver. Also, all the values must be “truly random” which is referred to as seed. This seed must be generated from the truly random sources, because it is utilized by GA techniques in order to generate larger sized keys. As shown in Figure 1, the seed is first processed by one of the existing statistical sound generators, namely, LCG. Through feedback mechanisms, the initial pad equal to the size of the plaintext is generated. That is, X_{-1} is used to generate X_0, X_0 is used to generate X_1, and so on, where for each computation the remaining secret key parameters, that is, the multiplier m, the increment c, and the modulus M, remain unchanged. The initial pad $X = (X_0, X_1, \ldots, X_n)$ is then converted into a population of individuals $Popu = (Popu_0, Popu_1, \ldots, Popu_n)$, where Popu_0 is a binary equivalent of integer X_0, $Popu_1$ is a binary equivalent of integer X_1, and so on. Afterward, the population is evolved by three evolutionary operators: selection, crossover, and mutation (all these operators have been discussed in detail

(1) **Data**: $KEY = X_{-1}, m, c, M, P_{cm}, LeastGen$.
(2) **Result**: one-time pad of size N, where N is the size of the plaintext.
Remark 1: Generate an initial population *Popu* of size N, where ith chromosome is represented as binary string equivalent to X_i. Here the initial vector $X = X_0, X_1, \ldots, X_n$ is computed via LCG equation, where $n = N - 1$. That is:
(3) **for** i from 0 to n **do**
(4) Set $X_i \leftarrow (X_{i-1} * m + c) \bmod (M)$
(5) **end for**
Remark 2: Initialize two positive integers: Number of selection of chromosome pairs for crossover *(Nopc)* and Number of selection of chromosomes for mutation *(Nom)* by "1". Sometime, we refer *Nopc* and *Nom* as crossover and mutation rates, respectively.
(6) Set $Nopc \leftarrow 1$, $Nom \leftarrow 1$
Remark 3: Update crossover and mutation rate, if the size of population is more than or equal to 10. That is:
(7) **if** $N \geq 10$ **then**
(8) Compute $Nopc \leftarrow \lceil P_{cm} * N \rceil$
(9) **if** $P_{cm} > 0.3$ **then**
(10) Compute $Nom \leftarrow \lfloor (P_{cm} * N)/2 \rfloor$. To maintain low probability mutation in case of large sized population.
(11) **else**
(12) Compute $Nom \leftarrow \lceil P_{cm} * N \rceil$
(13) **end if**
(14) **end if**
(15) Initialize $\pi_1 \leftarrow (X_n * m + c) \bmod (M)$
(16) $Generation \leftarrow 1$
(17) **repeat**
Remark 4: Select a pair of chromosome and determine a point of crossover. Repeat the process till counter reached *Nopc*.
(18) **for** i from 1 to *Nopc* **do**
(19) Set $index1st\ chromosome \leftarrow \pi_1 \pmod{N}$
(20) Compute $\pi_1 \leftarrow (m\pi_1 + c) \bmod (M)$
(21) Set $index2nd\ chromosome \leftarrow \pi_1 \pmod{N}$
(22) Compute $\pi_1 \leftarrow (m\pi_1 + c) \bmod (M)$
(23) Set $C_{\text{point}} \leftarrow |X_{index1st\ chromosome} - X_{index2nd\ chromosome}| \bmod (N')$, where C_{point} denotes the starting bit position of the chosen chromosomes to be reproduced. Here $N' = 8$, since a maximum integer value equivalent to its binary representation could be 255 (i.e., $2^8 - 1$).
(24) Perform single point crossover between pairs of selected chromosomes. That is between $\text{Popu}_{index1st\ chromosome}$ and $\text{Popu}_{index2nd\ chromosome}$ with probability P_{cm}, where the starting point for mating is denoted by C_{point}.
(25) **end for**
(26) Set $\pi_2 \leftarrow \pi_1$
Remark 5: Select an individual from the mated population and perform a single bit complement mutation for each of the following iteration, where the new population was generated in the previous steps (18)–(25).
(27) **for** i from 1 to *Nom* **do**
(28) Compute $\pi_2 \leftarrow (m\pi_2 + c) \bmod (M)$
(29) Set $indexSelectedChromosome \leftarrow \pi_2 \pmod{N}$
(30) Set $M_{\text{point}} \leftarrow X_{indexSelectedChromosome} \pmod{N'}$, where M_{point} denotes the position of a bit of the selected chromosome, $\text{Popu}_{indexSelectedChromosome}$.
(31) Change a bit of $\text{Popu}_{indexSelectedChromosome}$ located at position M_{point}.
(32) **end for**
(33) Update $Generation \leftarrow Generation + 1$
(34) Set $\pi_1 \leftarrow \pi_2$
(35) Update $\pi_1 \leftarrow (m\pi_1 + c) \bmod (M)$
(36) **until**{*Generation* counter reached *LeastGen*}

Algorithm 1: Pseudocode of the IGRKG method.

in Sections 5.1, 5.2, and 6). The probability of crossover and mutation is controlled by a common probability parameter P_{cm}. However, for each instance of the problem, the rate of mating and mutation may be different; we determine these rates by a deterministic mathematical procedure (for details, see Algorithm 1, Steps (6) to (14)). The selection of individuals for mating, crossover point C_{point}, choice of genes for mutation, and mutation point M_{point} are also controlled by a deterministic mathematical procedure (see Algorithm 1, Steps (19) to (23) and (28) to (30)). Finally, we get an obscure final pad $K = (k_0, k_1, \ldots, k_n)$, where k_i is an integer equivalent of the corresponding binary individual. The number of generations is controlled by a parameter, namely, *LeastGen* (for details about this parameter, see Section 5.2, Step (2)).

Advantages of IGRKG over GRKG. (1) IGRKG generator is much faster than the GRKG generator; for instance, in generating large sized secure pad (e.g., $N = 1000$), the average time

taken by the IGRKG generator is about 2.432 seconds, while the GRKG generator takes 9.747 seconds (for details, see Section 6.4). These results indicate that the IGRKG generator is four times faster than the GRKG generator. Consequently, in the case of exchange of a significant number of encrypted messages, the OTP-IGRKG system will outperform the OTP-GRKG system. (2) In terms of randomness, the quality of the IGRKG generator is significantly better than the GRKG generator (see statistical and randomness testing results in Section 6.5).

We organize the remainder of the paper as follows: in Section 3, we present some of the previous valuable research work in the field of cryptology where the genetic algorithm has been utilized. In Section 4, we present basics of the one-time pad and associated challenges. In Section 5, we propose the IGRKG method followed by comparison with previously proposed Sokouti et al. GRKG method. In Section 6, statistical testing and cryptanalysis results are discussed followed by conclusion in Section 7.

3. Genetic Algorithm

The origin of evolutionary algorithms (EAs) is an attempt to mimic some of the process taking place in natural evolution. Although the details of biological evolution are not completely understood (even nowadays), there exist some points supported by strong experimental evidence [5]. Genetic algorithm (GA) is one of the most popular EA techniques that has emerged based on the concept of imitating the evolution of a species [6]. In the case of GA, a population of individuals (or chromosomes) is generated using an intelligent method or a random method [6–8]. Each of these individuals is encoded as a binary string that represents a possible candidate solution to the problem at hand. In each iteration, the survival strength of each candidate solution is measured by a fitness function [6–9]. Afterward, the evolutionary process is constrained by three genetic operators: selection, crossover, and mutation. Through selection procedure, individuals are selected that enter into the crossover process. The crossover operator alters two or more parents to create offspring, where a probabilistic crossover rate is usually used to generate offspring [7, 8, 10]. Mutation operator produces one child from one parent by flipping a bit (s) of the parent. A probabilistic mutation rate is usually used to determine whether a particular change occurred or not within an individual [7, 8, 10].

There are some important characteristics of crossover and mutation operators that are not captured by the other. Błażej et al. [11] mentioned that it has never been theoretically shown that mutation is in some sense less powerful than the crossover and vice versa. Mutation serves to create random diversity in the population while crossover serves as an accelerator that promotes emergent behavior from components [11, 12]. The metaissue, then, is the relative importance of diversity and construction. It is impossible for mutation to simultaneously achieve high levels of construction and survival [11, 12]. This would appear to be important since one without the other may not be extremely useful. High construction levels are accomplished at the expense of survival (e.g., mutation rate 0.5), while good survival is at the expense of construction (e.g., mutation rate 0.01) [11, 12]. In our study, we get the highly constructive results with 0.25 to 0.3 mutation rates. That is, 25% to 30% parents are affected in our study by mutation operation (see Section 5.1 for details). GA parameters can be controlled in three different ways: deterministic [13, 14], adaptive [13–15], and self-adaptive [13, 14, 16]. The deterministic parameter control technique takes place when the value of strategy parameters (e.g., *Nopc* and *Nom* in our study) is altered by some deterministic rule. This rule modifies the strategy parameters deterministically without using any feedback from the search [13, 14, 17].

Applications of GAs in Cryptographic Applications. GAs have been applied successfully to solve real-world optimization and search problems [9]. These techniques have also shown good potential in the domain of cryptology. Here we mention some of the good works that have been carried out in the last decade. An interesting work in the domain of cryptographic protocol design has been carried out by Park and Hong [18] and Zarza et al. [19] in 2005 and 2006, respectively. Wang et al. (2012) [20] have proposed a novel method based on the genetic algorithm and chaotic map for designing substitution boxes (S-boxes). Jhajharia et al. (2013) [21] have utilized GAs for cryptographic key generation. Jain and Chaudhari (2014) [22] have proposed an improved GA method to attack the knapsack based cryptosystems. Faraoun (2014) [23] has proposed a block cipher design using GA and cellular automata. Recently, GA and CGP techniques have been utilized in [24] for determining strong cryptographic Boolean functions. Jain and Chaudhari (2015) [25] have proposed improved GA for automated cryptanalysis of the substitution ciphers. In [3] Sokouti et al. (2013) have proposed a GA technique for automatically generating OTP keys that we improve in this paper.

4. One-Time Pad Cryptosystems

One-time pad cryptosystems are based on the concept of stream cipher. In stream cipher, a short secret key is used to generate a keystream (i.e., a string of bits) [1]. The keystream bits are XORed with the plaintext bits in the usual way to produce the ciphertext [1]. At the receiver end, the ciphertext is XORed with keystream to get the original plaintext [1]. However, in stream cipher, a keystream is generated from a short secret key [1]. Therefore, these ciphers can be compromised if not used carefully. The advantage of stream ciphers is that they are much faster in hardware and therefore mostly employed in resource-constrained devices. However, the original OTP is used in those applications where the primary objective is perfect security rather than speed [1]. The conventional OTP cryptosystem combines a plaintext sized key with the given plaintext code as modulo addition “26” and thereby generates the ciphertext. An example is shown in Table 1. The fact is that the plaintext message can consist of not only English alphabet, but also ASCII characters. Therefore, in this paper we consider that the encryption and decryption of plaintext will be done on “modulo 256” rather than “modulo 26.” As a result, each plaintext character will

Table 1: Encryption and decryption process of conventional OTP cipher using modular arithmetic 26. P: plaintext, P_c: plaintext code, KEY: secret key, K_c: key code, C: ciphertext, $C_c = (P_c + K_c) \bmod (26)$ and $C'_c = (C_c - K_c) \bmod (26)$. From C'_c, P'_c is obtained as follows: $P'_c = C'_c$ if C'_c is +ve; otherwise $P'_c = C'_c + 26$.

Encryption										
P	O	N	E	T	I	M	E	P	A	D
P_c	14	13	4	19	8	12	4	15	0	3
KEY	V	M	C	E	A	P	H	I	N	R
K_c	21	12	2	4	0	15	7	8	13	17
C_c	9	25	6	23	8	1	11	23	13	20
C	J	Z	G	X	I	B	L	X	N	U
Decryption										
C	J	Z	G	X	I	B	L	X	N	U
C_c	9	25	6	23	8	1	11	23	13	20
KEY	V	M	C	E	A	P	H	I	N	R
K_c	21	12	2	4	0	15	7	8	13	17
C'_c	−12	13	4	19	8	−14	4	15	0	3
P'_c	14	13	4	19	8	12	4	15	0	3
P	O	N	E	T	I	M	E	P	A	D

consist of 8 bits (i.e., each plaintext character will be in the range $[0, 255]$).

5. Genetic Algorithm for Generating OTP Keys

There are two main challenges for developing original OTP cryptosystem: (1) The OTP cryptosystem must generate a key of length equal to the length of the plaintext. (2) The key should be truly random for achieving perfect security. Plaintexts are variable sized and often their size is large. Therefore, it is impossible to generate a truly random key of the size of the plaintext.

An efficient option for solving this kind of problem is the utilization of pseudorandom key [1]. However, it is not trivial to generate pseudorandom key equal to the length of the plaintext. In this context, Sokouti et al. (2013) [3] have proposed the GRKG method. GRKG generator accepts a fixed size short secret key as an initial key and thereby generates the pseudorandom key K. Here, we point out each time a different key k $(k \in K)$ is generated.

We have two popular pseudorandom generator choices as a base generator: LCG and Mersenne Twister, because of the good statistical properties [26]. However, for cryptographic security only the use of a statistical sound generator is not sufficient. Therefore, we can employ either LCG or Mersenne Twister for generating initial pad and then genetic algorithm is used to improve the randomness of the initial pad. As a result, an obscure and appropriate OTP key is generated. In this research, we have decided to use LCG method because it is easy to implement and runs efficiently on computer hardware [26]. Most importantly, its use allows us to give a fair comparison between two methods, GRKG and IGRKG, since LCG has also been employed in the GRKG method.

5.1. IGRKG: The Proposed Method. Consider initial key $KEY = \{X_{-1}, m, c, M, P_{cm}, LeastGen\}$ which consists of LCG and GA parameters, and the details are as follows.

Parameters Related to LCG

M: the modulus $(M > 0)$

X_{-1}: an initial positive integer number for generating another integer number using (1), where $(0 \le X_{-1} < M)$

m: the multiplier $(0 < m < M)$

c: the increment $(0 \le c < M)$.

Parameters Related to GA

P_{cm}: combine probability of crossover and mutation

$Nopc$: number of selections of chromosome pairs for crossover

Nom: number of selections of chromosomes for mutation

$LeastGen$: minimum number of iterations to generate a sufficient secure OTP key

$$X_i \longleftarrow (X_{i-1} * m + c) \bmod (M). \quad (1)$$

Algorithm 1 (Description). Pseudocode for the proposed IGRKG method is shown in Algorithm 1. Input to the algorithm is KEY, where KEY is a secret key decided by communicating parties once. Using first four KEY elements, X_{-1}, m, c, and M, the initial pad $X = \{X_0, X_1, \ldots, X_n\}$ is generated via LCG method, where $n = N - 1$ and N is the size of the plaintext. The initial pad is then converted into its equivalent binary representation (see Remark 1).

GA Operators. The binary initial pad which is generated by LCG method is modified by applying selection, crossover, and mutation that are deterministically [14] controlled. That is, crossover and mutation will be not performed at random positions of individuals; rather positions are determined

Table 2: Genetic parameters for the IGRKG method.

Step number	Parameter/operator	Value/type
(1)	Population size (N)	Size of the plaintext
(2)	Crossover	Single point crossover
(3)	Mutation	Single bit mutation
(4)	Rate of crossover, that is, % of parents mated	50% if $10 < N \leq 150$ 60% if $N > 150$
(5)	Rate of mutation, that is, % of parents mutated	25% if $10 < N \leq 150$ 30% if $N > 150$
(6)	Selection method (a parent & survival EA deterministic selection strategy [4])	Deterministic (see Algorithm 1, Steps (19) to (23) and Steps (28) to (30))

using deterministic procedure. It is emphasized that the same secret key is possible at both ends iff identical evolutionary operations are applied. If we use fitness function then this constraint will be violated. Therefore, this work does not require any fitness function; however, for generating secure OTP keys intelligent selection, crossover and mutation operators have been designed.

Deciding Values of Nopc and Nom. If the initial pad length $N < 10$, then selection of one pair of chromosomes (i.e., $Nopc = 1$) and a single chromosome (i.e., $Nom = 1$) is sufficient for reproduction and mutation, respectively (see Remark 2). However, for the initial pad of size $N \geq 10$, *Nopc* and *Nom* are determined deterministically by utilizing P_{cm} and N (see Steps (7) to (14)). P_{cm} is a common probability parameter for crossover and mutation.

Fine-Tuning of Crossover and Mutation. We have tested certain type of mutation and crossover operators, but the best results have been obtained using simple mutation (which flips a selected bit) and single point crossover. In the literature it has also been shown that, among all the crossover operators, the most successful one is single point crossover [27]. A deterministic procedure is developed for deciding crossover and mutation points (see Steps (23) and (30), resp.). The number of chromosomes mutated is defined as fixed percentage of the total number of chromosomes (see Steps (10)–(12)).

Finding best combination of crossover rate and mutation rate is an important step in GA. In [28, 29], it is investigated that generally low mutation rates (0.01 to 0.1) and comparatively high crossover rates (0.5 to 0.7) perform very well. However, in [8], it is mentioned that the modern view of EAs admits that specific problem types require specific EA setups. Therefore, different crossover and mutation rates have been experimented to investigate their capability to find good solutions (the conditional optimal values of crossover and mutation rates are shown in Table 2). Note that there is no prior experimental work of this kind, so this work should be considered as a guideline for future research.

Use of π_1 and π_2 Variables. For the initialization of π_1, we use the last element X_n of the initial pad only once (see Step (15)). That is, in the evolutionary process, we will never use X_n again due to security reasons, but the GRKG method uses X_n more than once, which is one of the drawbacks of the GRKG method (see Table 3, Steps (6) and (7)). Steps (19) and (21) show that an integer variable π_1 is used to select a chromosome pair for mating, where *each time* possibly a different chromosome pair is mated (see Steps (23) and (24)). In each iteration the mating operation is performed *"Nopc"* times (see Remark 4 and Step (18)). Step (29) shows that another integer variable π_2 is used to select an individual for mutation, where *each time* possibly a different individual is mutated (see Steps (30) and (31)). For each iteration the mutation operation is performed *"Nom"* times (see Step (27)). By repetitive applications of mating and mutation, a new population is generated. During evolution of the population through crossover, variable π_1 is itself updated (see Steps (20) and (22)). Similarly, during evolution of the population through mutation, variable π_2 is itself updated (see Steps (28) and (35)). In both cases, the LCG method is used. In each iteration, after crossover and mutation, π_2 and π_1 are assigned the updated value of π_1 and π_2, respectively (see Steps (26) and (34)). This strategy has been introduced in this research for the purpose of generating robust and secure OTP key (for detailed information, see Section 5.2, Step (3)).

Use of LeastGen Variable. Until the termination condition is not satisfied, the new population is fed back in the evolutionary process. *LeastGen* is an integer variable that indicates the minimum number of generations till the pad is entered in the evolutionary process (see Section 5.2, Step (2)).

5.2. Comparison between GRKG and IGRKG Generators. In this section, we compare the proposed IGRKG method with Sokouti et al.'s GRKG method [3]. A table of comparison based on the features of both the generators is shown in Table 3. In this table, we have underlined the values of IGRKG features that are different form their GRKG counterparts. A detailed list of proposed improvements is as follows:

(1) Rather than the secret key of size "seven," IGRKG uses a short secret key of size "six." This is possible because the crossover and mutation probabilities have been combined in a single parameter P_{cm}. However, the algorithm is designed in such a way that the same

Table 3: A comparison between features of GRKG and IGRKG (PGU: previous generation updated, CGU: current generation updated).

Step number	Features	GRKG	IGRKG (our approach)
(1)	Size of secret key	7	6
(2)	Secret key parameters	$X_{-1}, m, c, M, P_c, P_m, MaxGen$	$X_{-1}, m, c, M, P_{cm}, LeastGen$
(3)	Population size (N)	Size of the plaintext	Size of the plaintext
(4)	Number of genes in each chromosome	8	8
(5)	Method for generating initial population	LCG method	LCG method
(6)	Initialization of parameter for evolution of first generation (via crossover)	$\pi_1 \longleftarrow (X_n * m + a) \bmod (M)$	$\pi_1 \longleftarrow (X_n * m + a) \bmod (M)$
(7)	Initialization of parameter for evolution of first generation (via mutation)	$\pi_2 \longleftarrow (X_n * m + a) \bmod (M)$	$\pi_2 \longleftarrow \mathrm{CGU}(\pi_1)$
(8)	Determination of number of chromosome pairs for crossover *(Nopc)*	$Nopc \leftarrow P_c * N$ (no default setting)	$Nopc \leftarrow \lceil P_{cm} \times N \rceil$ (by default: $Nopc \coloneqq 1$)
(9)	Determination of number of chromosomes for mutation *(Nom)*	$Nom \leftarrow P_c * N$ (no default setting)	If $(P_{cm} > 0.3)$ then $Nom \leftarrow \lfloor (P_{cm} * N)/2 \rfloor$ Otherwise $Nom \leftarrow \lceil P_{cm} * N \rceil$ (by default: $Nom \coloneqq 1$)
(10)	Selection of index of chromosomes pairs for reproduction *(index1st chromosome* and *index2nd chromosome)*	Set $\pi_1 \leftarrow \mathrm{PGU}(\pi_1)$ and update it as $(\pi_1 * m + a) \bmod (M)$ Then $\pi_1 \pmod N$	Set $\pi_1 \leftarrow \mathrm{PGU}(\pi_2)$ and update it as $(\pi_1 * m + a) \bmod (M)$ Then $\pi_1 \pmod N$
(11)	Selection of index of chromosomes for mutation *(indexSelectedChromosome)*	Set $\pi_2 \leftarrow \mathrm{PGU}(\pi_1)$ and update it as $(\pi_2 * m + a) \bmod (M)$ Then $\pi_2 \pmod N$	Set $\pi_2 \leftarrow \mathrm{CGU}(\pi_1)$ and update it as $(\pi_2 * m + a) \bmod (M)$ Then $\pi_2 \pmod N$
(12)	Crossover mode	Single point	Single point
(13)	Computation of Crossover point (C_{point})	$(X_{index1st\ chromosome} + X_{index2nd\ chromosome}) \pmod 8$	$(\lvert X_{index1st\ chromosome} - X_{index2nd\ chromosome} \rvert) \pmod 8$
(14)	Mutation mode	Single bit complement	Single bit with low probability
(15)	Computation of mutation bit position (M_{point})	$X_{indexSelectedChromosome} \pmod 8$	$X_{indexSelectedChromosome} \pmod 8$
(16)	*(mean performance time) if* $N = 1000$, $LeastGen = MaxGen = 1000$	*9.747 seconds*	*2.432 seconds*
	Comparison of encryption and decryption function		
(17)	Encryption function (obtain ciphertext)	$Q_i = \dfrac{K_i}{P_i}$, $R = K_i \pmod{P_i}$ $C_i = \{Q_i, R_i\}$	$C_i = P_i \oplus K_i$
(18)	Decryption function (recover plaintext)	$P_i = \dfrac{(K_i - R_i)}{Q_i}$	$P_i = C_i \oplus K_i$

probability parameter is utilized for performing both crossover and mutation operations (see Table 3, steps (1) and (2)).

(2) Unlike *MaxGen* parameter used in GRKG, IGRKG uses *LeastGen* parameter. The essence of the *LeastGen* parameter is the minimum generations in which the initial pad is obscured almost completely. The IGRKG scheme has been tested with different values of P_{cm} and N. It is observed that 50 generations are sufficient to completely obscure the initial pad. However, after each communication the variable *LeastGen* is increased in order to achieve computationally high security.

(3) For evolving the existing generation, a different approach is proposed which is based on the effective updates of π_1 and π_2. In the GRKG method, the

same initial value (i.e., PGU(π_1)) is used for evolving current generation (see Table 3: Column 2, Steps (10) and (11)). The limitation of this approach is that the *Nom* number of individuals that were improved by crossover is once again selected for mutation. Due to this reason, the GRKG method requires a large number of generations for evolving the remaining individuals. This limitation is resolved by assigning the updated value of π_2 to π_1 and by assigning updated value of π_1 to π_2 (i.e., $\pi_1 \leftarrow$ PGU(π_2) and $\pi_2 \leftarrow$ CGU(π_1), see Table 3, Column 3, Steps (10) and (11), resp.). This phenomenon gives the chance to remaining individuals that were not improved by crossover, that is, improvement in the same current generation through mutation. The main benefit of this approach is that there is a high probability of selection of chromosomes for mating that were not selected in mutation and vice versa. This idea increases randomness of the pad with the increase in iteration. That is, due to this strategy the IGRKG method produces the more randomized pad in less number of iterations as compared to the GRKG method.

(4) In order to determine crossover points, GRKG method uses modular arithmetic over addition. This approach makes the GRKG scheme conceptually weak. The fact is that the sum of two chromosome values (i.e., sum of integers) before and after crossover will always be the same. That is, if two chromosomes A and B are mated and converted into A' and B', respectively, whenever in the next generation A' and B' are selected for crossover, the result will be the original chromosomes, that is, again A and B (see Table 3, Step (13)). Clearly, this phenomenon is a big obstacle in increasing randomness of the input pad. In this paper, we resolve this weakness by suggesting the use of modular arithmetic over subtraction rather than addition. Due to this strategy, even though the same pair will be selected in the next generation, the different crossover points will be selected because the subtraction of two chromosome values before and after crossover operation is different. This approach improves the practical efficiency of the generator.

(5) We have critically examined that the encryption and decryption functions suggested by Sokouti et al. [3] are not appropriate for use in cryptography. The design of encryption and decryption functions is not a part of the OTP key generator. However, as a complete OTP scheme, we advise simple encryption and decryption functions that are often used in stream ciphers (see Table 3, Steps (17) and (18)).

6. Results

For the purpose of comparison between GRKG and IGRKG generators in terms of speed, we have implemented both generators in Java 2.0 with Intel Quad-Core processor i7 (@3.40 Ghz). We present the results of both the generators on the text "cryptology." The size of plaintext "cryptology" is 10; that is, $N = 10$. Consider $P_{cm} = P_c = P_m = 0.2$. That is, in each iteration "two pair ($Nopc \leftarrow \lceil 0.2 * 10 \rceil = 2$)" of chromosomes and "two ($Nom \leftarrow \lceil (0.2 * 10) \rceil = 2$)" chromosomes will be affected by the crossover and mutation, respectively. Note that this example has been considered by Sokouti et al. [3] in their work. Therefore, for a fair comparison between GRKG and IGRKG generators, we demonstrate our work on the same example.

6.1. Common Computation. Consider secret key = ($X_{-1} = 9$, $m = 5$, $c = 7$, $M = 256$, $P_c = P_m = P_{cm} = 0.2$, $LeastGen = MaxGen = 10$). Using this short secret key an initial pad is generated iteratively via LCG method. That is, $X_0 = (9 * 5 + 7) \bmod (256) = 52$, $X_1 = (52 * 5 + 7) \bmod (256) = 11$, $X_2 = (11 * 5 + 7) \bmod (256) = 62$, and so on. Finally, $X = \{52, 11, 62, 61, 56, 31, 162, 49, 252, 243\}$. A population of size 10 is initialized, where ith chromosome will be binary equivalent of the ith element of X. This population is input in the GRKG and IGRKG generators for generating OTP keys.

6.2. Results Obtained Using the GRKG Method. In this section, we determine the OTP key from the initial pad using the GRKG generator, where the initial pad = $\{52, 11, 62, 61, 56, 31, 162, 49, 252, 243\}$. Table 4 shows the working of the GRKG method. Initially $\pi_1 = 243$, that is, the last element of the initial pad.

Mating. Initially, for mating, 8th and 9th chromosome pairs are selected, where the selection of chromosomes is determined as follows: (π_1, i.e., $243 * 5 + 7$) mod (256) = 198 (mod) 10 = 8 and (updated π_1, i.e., $198 * 5 + 7$) mod (256) = 229 (mod) 10 = 9. The mating is performed in between 8th and 9th indexed chromosomes at the 7th indexed-gene position. The index is computed as follows: (X_8 i.e., 252 + X_9 i.e., 243) mod (N' i.e., 8) = 7. Similarly, the second mating operation is performed in between 0th and 5th indexed chromosomes, where the mating starts from 3rd "(52 + 31) mod (8) = 3" indexed-gene position. Following such selection and crossover mechanisms, the initial pad $\{52, 11, 62, 61, 56, 31, 162, 49, 252, 243\}$ is transformed into $\{28, 11, 62, 61, 56, 55, 162, 49, 252, 243\}$.

Mutation. The mutation operation is performed using variable $\pi_2 = 243$ (here, we point out that the initial value 243 is used again for the selection of chromosomes for mutation, which is one of the drawbacks of the GRKG method). As shown in Table 4, the first mutation operation changes 4th "252 (mod) 8 = 4" indexed bit of the 8th indexed chromosome and the second mutation changes 3rd "243 (mod) 8 = 3" indexed bit of the 9th indexed chromosome. In this way, after first iteration, the intermediate pad $\{28, 11, 62, 61, 56, 55, 162, 49, 252, 243\}$ is transformed into $\{28, 11, 62, 61, 56, 55, 162, 49, 236, 251\}$.

Table 4 also shows the recomputation (Re) phase which is one of the limitations of the GRKG method, where recomputation appeared due to the selection of the same chromosome again (i.e., 8th one). Similarly, during the mutation operation, if the same chromosome is selected again, then the GRKG method performs the recomputation. Here

Table 4: Working of the GRKG method (U: updated, SPE: selected pad element).

$m = 5$, $c = 7$, $M = 256$, and initially $\pi_1 = 243$ Update π_1 as $(\pi_1 * 5 + 7) \bmod (256)$ Element selection: π_1 mod 10. Compute C_{point}	$U(\pi_1)$	Pad state	Index 7	6	5	4	3	2	1	0	SPE
GRKG: transformation of an initial pad to an obscure pad											
		Initial pad: (52, 11, 62, 61, 56, 31, 162, 49, 252, 243)									
$(243 * 5 + 7) \bmod (256) = 198 \pmod{10} = 8$	198	Before crossover	1	1	1	1	1	1	0	0	252
$(198 * 5 + 7) \bmod (256) = 229 \pmod{10} = 9$	229		1	1	1	1	0	0	1	1	243
$C_{\text{point}} = (X_8 + X_9) \bmod (N') = (252 + 243) \bmod (8) = 7$		After crossover	1	1	1	1	1	1	0	0	252
That is, starting point for mating is 7			1	1	1	1	0	0	1	1	243
		(52, 11, 62, 61, 56, 31, 162, 49, 252, 243)									
$(229 * 5 + 7) \bmod (256) = 128 \pmod{10} = 8$	128	Before crossover	0	0	1	1	0	1	0	0	52
Re: $(8 + 1) \bmod (10) = 9$, $(9 + 1) \bmod (10) = 0$	(Re: 0)		0	0	0	1	1	1	1	1	31
$(128 * 5 + 7) \bmod (256) = 135 \pmod{10} = 5$	135	After crossover	0	0	0	1	1	1	0	0	28
$C_{\text{point}} = (X_0 + X_5) \bmod (N') = (52 + 31) \bmod (8) = 3$			0	0	1	1	0	1	1	1	55
Initially $\pi_2 = 243$. Compute M_{point} Update π_2 as $5\pi_2 + 7 \pmod{256}$ Element selection: $\pi_2 \pmod{10}$	$U(\pi_2)$	(28, 11, 62, 61, 56, 55, 162, 49, 252, 243)									
$(243 * 5 + 7) \bmod (256) = 198 \pmod{10} = 8$	198	Before mutation	1	1	1	1	1	1	0	0	252
$M_{\text{point}} = (X_8) \bmod (N') = (252) \bmod (8) = 4$		After mutation	1	1	1	0	1	1	0	0	236
		(28, 11, 62, 61, 56, 55, 162, 49, 236, 243)									
$(198 * 5 + 7) \bmod (256) = 229 \pmod{10} = 9$	229	Before mutation	1	1	1	1	0	0	1	1	243
$M_{\text{point}} = (X_9) \bmod (N') = (243) \bmod (8) = 3$		After mutation	1	1	1	1	1	0	1	1	251
		After first generation: (28, 11, 62, 61, 56, 55, 162, 49, 236, 251)									

Repetition of the above process iteratively (eight more times) results in the following pad state along with updated π_1:

2nd-generation input: $\{\pi_1 \leftarrow 135\}$ and {Pad: (28, 11, 62, 61, 56, 55, 162, 49, 236, 251)}
3rd-generation input: $\{\pi_1 \leftarrow 219\}$ and {Pad: (140, 171, 62, 61, 56, 55, 2, 49, 236, 115)}
4th-generation input: $\{\pi_1 \leftarrow 239\}$ and {Pad: (12, 163, 62, 61, 56, 55, 2, 49, 252, 243)}
5th-generation input: $\{\pi_1 \leftarrow 195\}$ and {Pad: (52, 163, 62, 61, 56, 15, 2, 49, 236, 251)}
6th-generation input: $\{\pi_1 \leftarrow 87\}$ and {Pad: (52, 163, 62, 29, 57, 15, 50, 1, 236, 251)}
7th-generation input: $\{\pi_1 \leftarrow 171\}$ and {Pad: (52, 51, 62, 29, 169, 15, 246, 1, 236, 51)}
8th-generation input: $\{\pi_1 \leftarrow 191\}$ and {Pad: (52, 171, 238, 29, 59, 15, 246, 1, 60, 51)}
9th-generation input: $\{\pi_1 \leftarrow 147\}$ and {Pad: (52, 171, 238, 29, 51, 15, 246, 33, 28, 59)}

Pad state and updated π_1 after 9th generation: $\{\pi_1 \leftarrow 39\}$ and {Pad: (12, 43, 238, 21, 51, 15, 246, 33, 28, 187)}

Performance time: 1.918 msec.

we emphasize that this phenomenon needs improvement because, in the case of large sized plaintext, the efficiency of the scheme will degrade. This paper resolves this issue by removing recomputation phase and keeping updating the resulting pad through crossover, where the crossover is performed using modular subtraction rather than modular addition.

6.3. Results Obtained Using the IGRKG Method. In this section, we determine the OTP key from the initial pad using the IGRKG generator, where the initial pad is equal to {52, 11, 62, 61, 56, 31, 162, 49, 252, 243}. Table 5 shows the working of the IGRKG method. Initially $\pi_1 = 243$, that is, the last element of the initial pad.

Mating. First of all, 8th and 9th indexed chromosome pairs are selected for mating. The mating is started from 1st "$|252 - 243| \pmod{8} = 1$" indexed-gene. The second matoperation is performed in between 8th and 5th indexed chromosome pairs of the output pad generated in the previous step, where the mating starts from 3rd "$(52 + 31) \bmod (8) = 3$" indexed-gene positions. Following such selection and crossover mechanisms, the initial pad {52, 11, 62, 61, 56, 31, 162, 49, 252, 243} is converted into {52, 11, 62, 61, 56, 247, 162, 49, 26, 253}.

Mutation. The mutation operation is performed using variable $\pi_2 = 135$ (*i.e., updated* π_1). As shown in Table 5, the 8th indexed chromosome is mutated at 4th "252 (mod) 8 = 4" indexed-gene position and 9th indexed

Table 5: Working of the IGRKG method (U: updated, SPE: selected pad element).

Initially $\pi_1 = 243$. Compute C_{point}											
Update π_1 as $5\pi_1 + 7$ (mod) 256			Index								
Element selection: π_1 (mod) 10	U(π_1)	Pad state	7	6	5	4	3	2	1	0	SPE
IGRKG: efficient transformation of an initial pad to a more-obscured pad											
		Initial pad:									
		(52, 11, 62, 61, 56, 31, 162, 49, 252, 243)									
$(243 * 5 + 7)$ mod (256) = 198 (mod) 10 = 8	198	Before crossover	1	1	1	1	1	1	0	0	252
$(198 * 5 + 7)$ mod (256) = 229 (mod) 10 = 9	229		1	1	1	1	0	0	1	1	243
$C_{\text{point}} = \lvert X_8 - X_9 \rvert \bmod (N') = \lvert 252 - 243 \rvert \bmod (8) = 1$		After crossover	1	1	1	1	0	0	1	0	242
That is, starting point for mating is 1			1	1	1	1	1	1	0	1	253
		(52, 11, 62, 61, 56, 31, 162, 49, 242, 253)									
$(229 * 5 + 7)$ mod (256) = 128 (mod) 10 = 8	128	Before crossover	1	1	1	1	0	0	1	0	242
$(128 * 5 + 7)$ mod (256) = 135 (mod) 10 = 5	135		0	0	0	1	1	1	1	1	31
$C_{\text{point}} = \lvert X_8 - X_5 \rvert \bmod (N') = \lvert 252 - 31 \rvert \bmod (8) = 3$		After crossover	0	0	0	1	1	0	1	0	26
			1	1	1	1	0	1	1	1	247
Initially $\pi_2 = 135$. Compute M_{point}											
Update π_2 as $5\pi_2 + 7$ (mod) 256											
Element selection: π_2 (mod) 10	U(π_2)	(52, 11, 62, 61, 56, 247, 162, 49, 26, 253)									
$(135 * 5 + 7)$ mod (256) = 170 (mod) 10 = 0	170	Before mutation	0	0	1	1	0	1	0	0	52
$M_{\text{point}} = (X_0) \bmod (N') = (52) \bmod (8) = 4$		After mutation	0	0	1	0	0	1	0	0	36
		(36, 11, 62, 61, 56, 247, 162, 49, 26, 253)									
$(170 * 5 + 7)$ mod (256) = 89 (mod) 10 = 9	89	Before mutation	1	1	1	1	1	1	0	1	253
$M_{\text{point}} = (X_9) \bmod (N') = (253) \bmod (8) = 5$		After mutation	1	1	0	1	1	1	0	1	221
		After first generation:									
		(36, 11, 62, 61, 56, 247, 162, 49, 26, 221)									
Repetition of the above process iteratively (six more times) results in the following pad state along with updated π_1:											
2nd-generation input: $\{\pi_1 \leftarrow 89\}$ and {Pad: (36, 11, 62, 61, 56, 247, 162, 49, 26, 221)}											
3rd-generation input: $\{\pi_1 \leftarrow 239\}$ and {Pad: (52, 11, 62, 61, 56, 247, 218, 49, 26, 133)}											
4th-generation input: $\{\pi_1 \leftarrow 53\}$ and {Pad: (244, 11, 62, 29, 57, 55, 218, 49, 130, 29)}											
5th-generation input: $\{\pi_1 \leftarrow 171\}$ and {Pad: (244, 3, 62, 29, 59, 55, 24, 219, 130, 53)}											
6th-generation input: $\{\pi_1 \leftarrow 209\}$ and {Pad: (244, 187, 62, 29, 11, 55, 24, 219, 2, 21)}											
7th-generation input: $\{\pi_1 \leftarrow 39\}$ and {Pad: (100, 187, 62, 157, 11, 55, 24, 3, 218, 53)}											
Pad state and updated π_1 after 7th generation: $\{\pi_1 \leftarrow 45\}$ and {Pad: (116, 187, 52, 221, 11, 183, 24, 3, 154, 63)}											
Performance time: 0.743 msec.											

chromosome is mutated at 3rd "243 (mod) 8 = 3" indexed-gene position. In this way, the intermediate pad {52, 11, 62, 61, 56, 247, 162, 49, 26, 253} is converted into {36, 11, 62, 61, 56, 247, 162, 49, 26, 221}.

6.4. Discussion on the Speed of GRKG and IGRKG Generators. It is observed from Tables 4 and 5 that the OTP key is generated by GRKG and IGRKG in "nine" and "seven" iterations, respectively. That is, the IGRKG method obscures the initial pad in less number of generations than the GRKG method. In other words, if we run the IGRKG generator for two more generations, it will result in the enhancement of the randomness. As evident from the results, the relative time performance of the IGRKG method is significantly better than the GRKG method. The time taken by the GRKG method and the IGRKG method for the above-solved instance is 1.918 and 0.743 milliseconds, respectively. For an accurate comparison between the speeds, we have tested both the generators on the large data set.

We have taken $M = 256$ and $P_{cm} = P_c = P_m = 0.2$ so that $Nopc = Nom = 2$ and $LeastGen = MaxGen = 1000$. Afterward, both the generators have been run 100 times for the plaintext of length of 1000 characters along with various settings of (X_{-1}, m, c). We have examined that the average time taken by the IGRKG method for 1000 generation is 2.432 seconds, while the GRKG method takes 9.747 seconds. This result indicates that the IGRKG generator is approximately four times faster than the GRKG generator.

6.5. Statistical Results. This section presents some statistical tests to analyze the security of GRKG and IGRKG generators that are purported to be random bit generators. A random bit generator is a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits. According to [1], it is impossible to give a mathematical

proof that a generator is a random bit generator; the tests described here help to detect certain kinds of weaknesses the generators may have. From both the generators, we take a sample output binary sequence s of length $n = 20000$ bits and subject it to four statistical tests designated as $Test_1$, $Test_2$, $Test_3$, and $Test_4$. The conclusion of each test is not definite but rather probabilistic. If the sequence passes all of the statistical tests, the generator is "not rejected" [1]. Here, we emphasize that normal and χ^2 distribution (*goodness of fit* test) tests can be used to compare the observed frequencies of a sequence to their expected frequencies under a hypothesized distribution. The χ^2 distribution with v degrees of freedom arises in practice when the squares of v independent random variables having standard normal distributions are summed. For detailed information on normal and χ^2 distributions, we refer readers to [1]. In the following sections, we present the formal definition of each of the four statistical tests and the results of statistical testing on the output sequence s.

6.5.1. Frequency Test. The objective of this test is to examine whether the numbers of 0s and 1s in s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0s and 1s in s, respectively. We use these statistics as [1]:

$$Test_1 = \frac{(n_0 - n_1)^2}{n}, \tag{2}$$

which approximately follows an χ^2 distribution with 1 degree of freedom.

6.5.2. Serial Test. The objective of this test is to examine whether the numbers of occurrences of 00, 01, 10, and 11 as subsequences of s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0s and 1s in s, respectively, and let n_{00}, n_{01}, n_{10}, and n_{11} denote the number of occurrences of 00, 01, 10, and 11 in s, respectively, where $n_{00} + n_{01} + n_{10} + n_{11} = n - 1$, since the subsequences are allowed to overlap. We use these statistics as [1]:

$$Test_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1, \tag{3}$$

which approximately follows an χ^2 distribution with 2 degrees of freedom.

6.5.3. Autocorrelation Test. The purpose of this test is to check for correlations between the sequence s and (noncyclic) shifted versions of it. Let d be a fixed integer, $1 \le d \le \lfloor n/2 \rfloor$. The number of bits in s not equal to their d-shifts is $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$. We use these statistics as [1]:

$$Test_3 = \frac{2(A(d) - (n-d)/2)}{\sqrt{n-d}} \tag{4}$$

which approximately follows a normal distribution with mean of zero and standard deviation of 1 if $n - d \ge 10$.

6.5.4. Poker Test. Let r be a positive integer such that $\lfloor n/m \rfloor \ge 5 \cdot (2^r)$, and let $k = \lfloor n/m \rfloor$. Divide the sequence s into k nonoverlapping parts each of length r, and let n_i be the number of occurrences of the ith type of sequence of length r, $1 \le i \le 2^r$. The poker test determines whether the sequences of length r each appear approximately the same number of times in s, as would be expected for a random sequence. We use these statistics as [1], which approximately follows an χ^2 distribution with $2^r - 1$ degrees of freedom:

$$Test_4 = \frac{2^r}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k. \tag{5}$$

For a significance level of $\alpha = 0.05$, threshold value for $Test_1 = 3.8415$, $Test_2 = 5.9915$, $Test_3 = 1.6449$ and $Test_4$ value is different for different degree of freedom which is computed using different size subsequences (see Table 10). The calculated $Test_j$ ($1 \le j \le 4$) results clearly indicate that the statistical results obtained via the IGRKG generator are consistently superior to the GRKG generator (see Tables 7–10).

In addition to the above-discussed statistical tests, we have tested large output sequence s of both the generators on the more stringent batteries of statistical test: Diehard, NST, and "ENT" [30]. For this purpose, we have generated two separate files of 250 MB corresponding to the output of each of the generators over a low entropy input, and then each file was analyzed with each of the batteries. Note that 250 MB data is a vast binary sequence s generated by using input parameters KEY_1 to KEY_4. Table 11 shows the results of the ENT test. As evident from the ENT results, the output of the GRKG and the IGRKG generators successfully passes all the tests. However, it is clear from the results that the ENT test results are superior in the case of the proposed IGRKG scheme. Diehard results are shown in Table 12. The tests are treated as successful if p value is greater than 0.05. IGRKG passes all the tests with significantly better results than GRKG. It is important to note that GRKG has failed in monkey test OPSO and overlapping sums test.

In the case of NIST test, 100 p values have been evaluated for each test; the proportion of successful results are presented in Table 13. The tests are treated as successful if p value is greater than 0.959 (except discrete Fourier transformation and binary matrix ranks $6 * 8$ tests). In these two tests the p value should be in between 0.051 and 0.990 for success [31]. Results presented in Table 13 show that the IGRKG generator passes all the tests; however, the GRKG generator has failed in longest-run test and binary matrix ranks $31 * 31$ and $32 * 32$ tests.

6.6. GRKG and IGRKG Quality Assessment

6.6.1. Diehard Scores. Although GRKG and IGRKG generators have been developed and reported in the literature for OTP key generation, we compare the performance of both the generators with several other existing pseudorandom number generators. The generators that we are comparing to GRKG and IGRKG are of various types: pure linear congruential generators (rand [32], rand1k [33], and pm

Table 6: Parameter values corresponding to (X_{-1}, m, c, M, P_{cm}, $LeastGen$) for generating four different 20000-bit binary sequences to be tested for examining randomness property of two generators: GRKG and IGRKG.

Step number	Parameter name	Parameter values	Remark
(1)	KEY_1	(9, 5, 7, 256, 0.5, 50)	For an initial value of *LeastGen*, 2000 bits are generated and then we increment *LeastGen* by one, for the incremented value again 2000 bits are generated, and so on. For instance, in the case of KEY_1 the last parameter is incremented up to 10 with step size +1, so that the last value will be 59. In this way, 20000 bits are generated, that is, a concatenation of 2000 bits ten times, where for each *KEY* a different 20000-bit-long sequence is generated.
(2)	KEY_2	(9, 5, 7, 256, 0.6, 60)	
(3)	KEY_3	(46, 9, 7, 256, 0.5, 50)	
(4)	KEY_4	(46, 9, 7, 256, 0.6, 60)	

Table 7: $Test_1$ values for a binary sequence s of size $n = 20000$, where s is generated individually from both the GRKG and the IGRKG generators corresponding to four different input *KEY* parameters that have been mentioned in Table 6.

Method name (n = size of s in bits)	Initial key parameters	Numbers of 0s and 1s in s		Computed $Test_1$ values	Threshold value at $\alpha = 0.05$ [1]
		n_0	n_1		
GRKG (n = 20000)	KEY_1	10089	9911	1.5842	3.8415
	KEY_2	9991	10009	0.0162	
	KEY_3	10040	9960	0.32	
	KEY_4	10047	9953	0.4418	
IGRKG (n = 20000)	KEY_1	10073	9934	*0.9660*	
	KEY_2	9993	10007	*0.0098*	
	KEY_3	10012	9988	*0.0288*	
	KEY_4	9986	10014	*0.0392*	

Table 8: $Test_2$ values for a binary sequence s of size $n = 20000$, where s is generated individually from both the GRKG and the IGRKG generators corresponding to four different input *KEY* parameters that have been mentioned in Table 6.

Method name (n = size of s in bits)	Initial key parameters	Number of 0s and 1s in s and number of occurrences of 00, 01, 10, and 11 in s						Computed $Test_2$ values	Threshold value at $\alpha = 0.05$ [1]
		n_0	n_1	n_{00}	n_{01}	n_{10}	n_{11}		
GRKG (n = 20000)	KEY_1	10089	9911	5051	5037	5056	4855	4.0422313	5.9915
	KEY_2	9991	10009	4961	5030	4995	5013	0.5067761	
	KEY_3	10040	9960	5064	4976	5040	4919	2.2466783	
	KEY_4	10047	9953	5024	5023	4906	5046	1.9696706	
IGRKG (n = 20000)	KEY_1	10073	9927	5066	5007	4993	4933	*0.7228394*	
	KEY_2	9993	10007	4979	5013	4995	5012	*0.1459578*	
	KEY_3	10012	9988	5037	4975	5014	4973	*0.5549792*	
	KEY_4	9986	10014	4972	5013	4996	5018	*0.2193629*	

[34]), multiply-with-carry generators (mother [35]), additive and subtractive generators (add [32], sub [34]), compound generators (shsub [32], shpm [34], and shlec [34]), feedback shift register generators (tgfsr [36], fsr [37]), Tausworthe generators (tauss [38]), and GP based generator (Lamar [31]). For the comparison purpose, Johnson's scoring method [39] is used. We have generated 50 different 10 MB files from GRKG and IGRKG using the same method as mentioned in Table 6, and then scores have been assigned using results of the Diehard tests. The score corresponding to different generators has been taken from [31]. Since Diehard tests produce one or more p values, categorizing them as rejected, suspect, or good, a p value is called rejected if $p \geq 0.998$ and suspect if $0.95 \leq p < 0.998$; all other p values are considered to be good. Two points, one point, and zero points have been assigned for rejection, suspect, and good, respectively. Finally, the addition of these points produces a global Diehard score for each generator. The average has been taken over the 50 evaluations in the case of GRKG and IGRKG generators. In Table 14, low scores indicate good quality generators. From Table 14, it can be noted that the IGRKG generator is comparatively better than Lamar and significantly superior to the rest of the generators.

6.6.2. Changes in Population. For the purpose of demonstration of behavior of the proposed IGRKG method, we present some experimental analysis. For instance, consider that the secret key equals ($X_{-1} = 46$, $m = 9$, $c = 7$, $M = 256$, $P_{cm} = 0.2$, $LeastGen = 10$) and size of the plaintext is 10. Figure 2 shows the changes in population for each iteration, where the initial pad (165, 212, 123, 90, 49, 192, 199, 6, 61, 44) is indicated by the 0th generation. Figures 2(a), 2(b), and 2(c)

Table 9: $Test_3$ values for a binary sequence s of size $n = 20000$, where s is generated individually from both the GRKG and the IGRKG generators corresponding to four different input KEY parameters that have been mentioned in Table 6.

Method name (n = size of s in bits)	Initial key parameter	For $d = 4000$ $A(d) :=$	Computed autocorrelation value	Threshold value (normal distribution) at $\alpha = 0.05$ [1]
GRKG ($n = 20000$)	KEY_1	7903	−0.485	1.6449
	KEY_2	8007	0.035	
	KEY_3	7922	−0.39	
	KEY_4	7971	−0.145	
IGRKG ($n = 20000$)	KEY_1	7897	*−0.515*	
	KEY_2	8001	*0.005*	
	KEY_3	7992	*−0.04*	
	KEY_4	8002	*0.01*	

Table 10: $Test_1$ values for a binary sequence s of size $n = 20000$, where s is generated individually from both the GRKG and the IGRKG generators corresponding to four different input KEY parameters that have been mentioned in Table 6.

Method (n = size of s in bits)	Subseqs. size r (in bits)	Number of subseqs. k	Deg. of freedom $(2^r - 1)$	Computed $Test_4$ values for different keys				Threshold value at $\alpha = 0.05$ [1]
				KEY_1	KEY_2	KEY_3	KEY_4	
GRKG ($n = 20000$)	4	5000	15	14.3217	12.5282	13.0562	11.7293	24.9958
	5	4000	31	31.8201	28.8756	28.6031	26.9173	44.9853
	7	2857	127	103.2013	88.2361	90.4128	85.1043	154.3015
	8	2500	255	211.921	193.2073	188.7217	189.6553	293.2478
IGRKG ($n = 20000$)	4	5000	15	*11.2961*	*10.2332*	*9.5024*	*8.1945*	
	5	4000	31	*23.191*	*19.9245*	*21.1387*	*24.2046*	
	7	2857	127	*82.1302*	*78.7643*	*72.9932*	*69.9462*	
	8	2500	255	*171.9173*	*142.108*	*138.3213*	*132.3265*	

Table 11: Results obtained by ENT batteries of statistical test (GRKG versus IGRKG).

Test	Result by GRKG	Result by IGRKG
Entropy	7.399413 bits/byte	*7.999999* bits/byte
Compression rate	0%	0%
Monte Carlo estimation	3.120513826 (error 0.1%)	*3.141524737* (error 0%)
χ^2 (goodness-of-fit)	217.14	*248.126*
Arithmetic mean	119.1214 (127.5: random)	*127.3974* (127.5: random)
Serial correlation coefficient	−0.00013 (uncorrelated = 0.0)	*−0.00007* (uncorrelated = 0.0)

represent the behavior of the algorithm for first three (0 to 2), second three (3 to 5), and the last three (6 to 8) generations, respectively, where 0th generation indicates the initial pad status and 8th generation represents the final pad status. The 1st and 2nd generations pad status can be seen in Figure 2(a) which are (37, 52, 115, 94, 209, 192, 199, 134, 61, 44) and (37, 52, 91, 126, 209, 192, 199, 198, 61, 44), respectively. The 3rd, 4th, and 5th generations pad status can been seen in Figure 2(b) which are (37, 52, 195, 126, 209, 193, 71, 94, 61, 44), (37, 52, 203, 126, 209, 195, 71, 94, 61, 44), and (37, 52, 195, 126, 209, 195, 47, 94, 61, 65), respectively. The 6th, 7th, and 8th generations pad status can been seen in Figure 2(c) which are (223, 36, 195, 126, 209, 195, 47, 36, 61, 65), (223, 36, 35, 222, 209, 195, 175, 52, 61, 65), and (95, 36, 53, 60, 209, 203, 175, 34, 223, 65), respectively. Although after 8th generation the initial pad is completely changed (see Figure 2(d)) and transforms to (95, 36, 53, 60, 209, 203, 175, 34, 223, 65), further iterations increase the randomness of the pad. Figure 2(d) shows a comparison graph of the initial and final population (output at the 8th generation), where we can clearly see that there is no similarity at any element positions of initial and final pads. In other words, initial and final pads are independent of each other; that is, without knowing the secret key, the initial pad is very difficult to recover.

Table 12: Results obtained using Diehard suite ($Test_5$: birthday spacings, $Test_6$: GCD, $Test_7$: overlapping permutations, $Test_8$: monkey test OPSO, $Test_9$: monkey test OQSO, $Test_{10}$: monkey test DNA, $Test_{11}$: count the 1s in a stream of bytes, $Test_{12}$: count the 1s in the specific bytes, $Test_{13}$: parking lot test, $Test_{14}$: minimum distance test, $Test_{15}$: random spheres test, $Test_{16}$: the squeeze test, $Test_{17}$: overlapping sums test, $Test_{18}$: runs-up-and-down test, $Test_{19}$: the Craps test).

Test	p value (GRKG)	p value (IGRKG)
$Test_5$	0.812	*0.912*
$Test_6$	0.702	*0.812*
$Test_7$	0.893	*0.982*
$Test_8$	**0.043**	*0.286*
$Test_9$	0.593	*0.673*
$Test_{10}$	0.692	*0.809*
$Test_{11}$	0.814	*0.903*
$Test_{12}$	0.778	*0.879*
Overall p value (GRKG)		0.312
$Test_{13}$	0.461	*0.513*
$Test_{14}$	0.683	*0.811*
$Test_{15}$	0.492	*0.686*
$Test_{16}$	0.738	*0.762*
$Test_{17}$	**0.047**	*0.252*
$Test_{18}$	0.356	*0.381*
$Test_{19}$	0.903	*0.987*
Overall p value (IGRKG)		0.373

6.7. Basic Cryptanalysis of GRKG and IGRKG. This section discusses resistance results against some of the basic cryptanalytic attacks that are as follows:

(1) *Input-Based Attack*. In order to distinguish between random outputs and GRKG or IGRKG outputs, if it is possible to use control or knowledge of the generator, then we say that the generator is not resistant to input-based attack.

(2) *State Compromise Extension Attack*. Assuming that a state $S1$ has been recovered by the adversary through successful efforts (e.g., due to inadvertent leak, a cryptanalytic success, etc.), an attack which tries to extend the advantage of state $S1$ is called state compromise extension (SCE) attack. SCE attack succeeds when the attacker is able to distinguish between random outputs and generator outputs before $S1$ was compromised. Due to insufficient starting entropy the generator can be started from an insecure guessable state; at that time there is a highest probability of SCE attack to work. SCE attack can also work when $S1$ has been compromised by any of the attacks mentioned below.

(a) *Backtracking Attack*. In order to acquire previous generator values, the backtracking attack uses the compromise of the state $S1$ at time t.

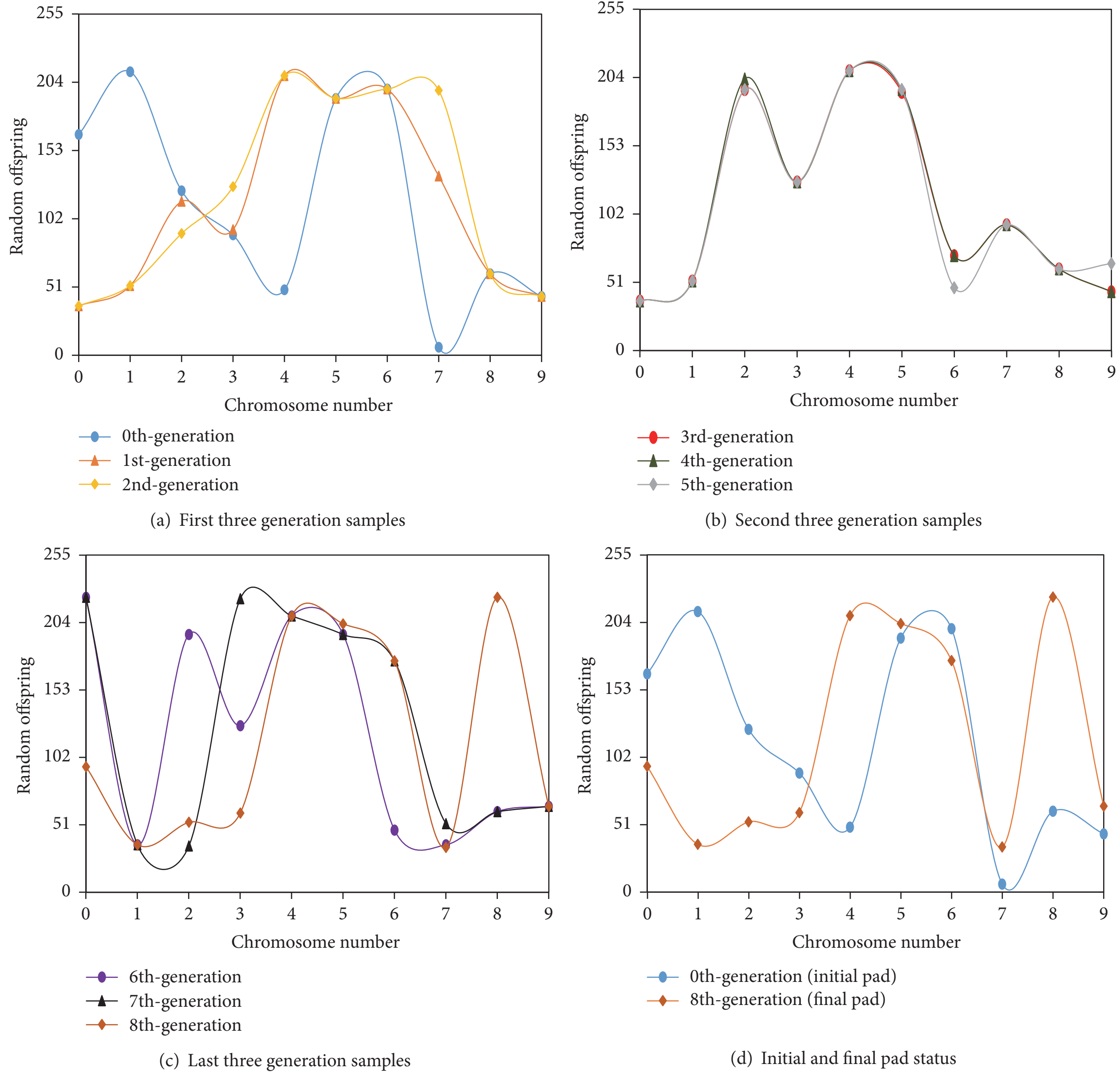
(b) *Permanent Compromise Attack*. As soon as the attacker negotiates $S1$ at time t, all past and

Table 13: Results obtained using NIST suite ($Test_{20}$: frequency, $Test_{21}$: block frequency, $Test_{22}$: cumulative sums, $Test_{23}$: runs, $Test_{24}$: longest run, $Test_{25}$: rank, $Test_{26}$: FFT, $Test_{27}$: overlapping templates, $Test_{28}$: nonoverlapping templates, $Test_{29}$: binary matrix ranks $31 * 31$, $Test_{30}$: binary matrix ranks $32 * 32$, $Test_{31}$: binary matrix ranks $6*8$, $Test_{32}$: discrete Fourier transform, $Test_{33}$: Maurer, $Test_{34}$: Apen, $Test_{35}$: linear complexity, $Test_{36}$: random Excursions, $Test_{37}$: random excursions variant, $Test_{38}$: serial).

Test	Proportion (GRKG)	Proportion (IGRKG)
$Test_{20}$	0.9700	*1.0000*
$Test_{21}$	0.9800	*1.0000*
$Test_{22}$	0.9800, 0.9800	*1.0000, 1.0000*
$Test_{23}$	1.0000	1.0000
$Test_{24}$	**0.9550**	*0.9800*
$Test_{25}$	1.0000	1.0000
$Test_{26}$	0.9700	*0.9950*
$Test_{27}$	0.9720	*1.0000*
$Test_{28}$	0.9680	*1.0000*
$Test_{29}$	**0.9520**	*0.9742*
$Test_{30}$	**0.9423**	*0.9659*
$Test_{31}$	0.2250	*0.4529*
$Test_{32}$	0.9732	*0.9900*
$Test_{33}$	0.1342	*0.4563*
$Test_{34}$	1.0000	1.0000
$Test_{35}$	0.9800	*1.0000*
$Test_{36}$	0.9732, 0.9732	*1.0000, 0.9822*
	0.9656, 0.9656	*0.9900, 0.9900*
	1.0000, 1.0000	1.0000, 1.0000
	1.0000, 1.0000	1.0000, 1.0000
$Test_{37}$	1.0000, 1.0000, 1.0000	1.0000, 1.0000, 1.0000
	0.9859, 0.9859, 1.0000	*1.0000, 1.0000, 1.0000*
	0.9718, 0.9718, 0.9718	*0.9859, 1.0000, 1.0000*
	1.0000, 1.0000, 1.0000	1.0000, 1.0000, 1.0000
	0.9656, 0.9656, 0.9718	*0.9859, 0.9859, 1.0000*
$Test_{38}$	0.9732, 0.9732	*0.9900, 0.9900*

Table 14: Pseudorandom number generators Diehard scores.

Pseudorandom number generators	Total score	Mean
rand1k	2129	66.53125
rand	9337	291.78125
pm	1619	50.59375
mother	602	18.8125
add	577	18.03125
sub	655	20.46875
shsub	548	17.125
shpm	799	24.96875
shlec	751	23.46875
fsr	573	17.90625
tgfsr	584	18.25
tauss	935	29.21875
lamar	377	11.78125
GRKG	578	18.46125
IGRKG	*374*	*11.74125*
True random variable	*371.072*	*11.596*



(a) First three generation samples

(b) Second three generation samples

(c) Last three generation samples

(d) Initial and final pad status

FIGURE 2: Changes in population with increase in generations (*LeastGen* = 10, P_{cm} = 0.2).

future $S1$ values are susceptible to attack which is a permanent compromise attack.

(c) *Iterative Guessing Attack*. If the inputs collected between times t and $t + \epsilon$ are guessable by the attacker, then this type of attack is called iterative guessing attack.

(d) *Meet-in-the-Middle Attack*. A combination of backtracking and iterative guessing attacks is called meet-in-the-middle attack. Knowledge of S at times t and $t + 2\epsilon$ allows the attacker to recover $S1$ at time $t + \epsilon$.

A binary sequence S of size 20000 bits is generated individually from both the GRKG and the IGRKG generators corresponding to four different input *KEY* parameters that have been mentioned in Table 6. Afterward, cryptanalysis against all the above-mentioned attacks has been performed. The cryptanalytic results have been mentioned in Table 15. Results obtained indicate that the IGRKG method is resistant to all the attacks; however, the GRKG method is not resistant to backtracking, iterative guessing, and meet-in-middle attacks.

7. Conclusion and Avenues for Future Research

This paper has presented an improved and efficient genetic-based OTP key generator. The proposed method is a significant improvement in the GRKG method. The proposed IGRKG generator has successfully passed the simple statistical tests such as frequency, serial, autocorrelation, and poker

Table 15: Cryptanalytic results.

Generator	Attacks				
	Input-based attack	Backtracking attack	Permanent compromise attack	Iterative guessing attack	Meet-in-the-middle attack
GRKG	Resistance	Nonresistance	Resistance	Nonresistance	Nonresistance
IGRKG	Resistance	Resistance	Resistance	Resistance	Resistance

tests. IGRKG generator has also passed ENT, Diehard, and NIST batteries of statistical tests. IGRKG is also resistant to basic cryptanalytic attacks. These tests indicate that IGRKG generator does not have any weakness and implementation bugs. Additionally, the statistical quality of the IGRKG generator has been compared with other existing pseudorandom number generators through Diehard scores, and the obtained scores indicate that IGRKG is the acceptable pseudorandom number generator. In terms of speed, IGRKG generator is four times faster than the GRKG generator.

It is important to note that, in the case of the IGRKG method, there are various trade-offs to run and produce the next pad. For instance, in the next communication, the *LeastGen* variable can be increased by "1." Another practical approach can be designed by an appropriate use of secret key parameters c and m, and that can be decided by communicating parties once. Indeed, if the *LeastGen* variable is correctly handled, this may result in computationally high security.

This paper has used linear congruential generator for generating initial pad and then genetic algorithm is used to improve the randomness of the initial pad. Instead of linear congruential generator, Mersenne Twister can also be used. However, how will the use of Mersenne Twister be effective and efficient in generation of OTP key? We left it as an open problem. Also, an extensive cryptanalysis is required to ensure computational security of the proposed generator.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2010.

[2] D. Tagu, J. K. Colbourne, and N. Nègre, "Genomic data integration for ecological and evolutionary traits in non-model organisms," *BMC Genomics*, vol. 15, no. 1, article no. 490, 2014.

[3] M. Sokouti, B. Sokouti, S. Pashazadeh, M.-R. Feizi-Derakhshi, and S. Haghipour, "Genetic-based random key generator (GRKG): A new method for generating more-random keys for one-time pad cryptosystem," *Neural Computing and Applications*, vol. 22, no. 7-8, pp. 1667–1675, 2013.

[4] K. A. De Jong, *Evolutionary Computation: A Unified Approach*, MIT press, 2006.

[5] S. Sivanandam and S. Deepa, *Introduction to Genetic Algorithms*, Springer Science & Business Media, 2007.

[6] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesly, 1989.

[7] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[8] Z. Michalewicz, *Genetic Algorithms+ Data Structures= Evolution Programs*, Springer Science & Business Media, 2013.

[9] T. F. Gonzalez, *Handbook of Approximation Algorithms And Metaheuristics*, CRC Press, 2007.

[10] M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*, vol. 12, MIT Press, 1999.

[11] P. Błażej, M. Wnętrzak, and P. Mackiewicz, "The role of crossover operator in evolutionary-based approach to the problem of genetic code optimization," *BioSystems*, vol. 150, pp. 61–72, 2016.

[12] W. M. Spears et al., *Crossover or Mutation, Foundations of Genetic Algorithms 2*, 1992.

[13] Á. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[14] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter Control in Evolutionary Algorithms: Trends and Challenges," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 167–187, 2015.

[15] M. Srinivas and L. M. Patnaik, "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 24, no. 4, pp. 656–667, 1994.

[16] J. E. Smith and T. C. Fogarty, "Adaptively parameterised evolutionary systems: Self adaptive recombination and mutation in a genetic algorithm," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1141, pp. 441–450, 1996.

[17] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 19–31, 2011.

[18] K. Park and C. Hong, "Cryptographic protocol design concept with genetic algorithms," in *Knowledge-Based Intelligent Information and Engineering Systems*, pp. 483–489, Springer, 2005.

[19] L. Zarza, J. Pegueroles, M. Soriano, and R. Martnez, "Design of cryptographic protocols by means of genetic algorithms techniques," in *SECRYPT*, pp. 316–319, 2006.

[20] Y. Wang, K.-W. Wong, C. Li, and Y. Li, "A novel method to design S-box based on chaotic map and genetic algorithm," *Physics Letters A: General, Atomic and Solid State Physics*, vol. 376, no. 6-7, pp. 827–833, 2012.

[21] S. Jhajharia, S. Mishra, and S. Bali, "Public key cryptography using neural networks and genetic algorithms," in *Proceedings of the 2013 6th International Conference on Contemporary Computing, IC3 2013*, pp. 137–142, Noida, India, August 2013.

[22] A. Jain and N. S. Chaudhari, "Cryptanalytic results on knapsack cryptosystem using binary particle swarm optimization," *Advances in Intelligent Systems and Computing*, vol. 299, pp. 375–384, 2014.

[23] K. M. Faraoun, "A genetic strategy to design cellular automata based block ciphers," *Expert Systems with Applications*, vol. 41, no. 17, pp. 7958–7967, 2014.

[24] A. Jain and N. S. Chaudhari, "Evolving highly nonlinear balanced boolean functions with improved resistance to DPA attacks," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9408, pp. 316–330, 2015.

[25] A. Jain and N. S. Chaudhari, "A new heuristic based on the cuckoo search for cryptanalysis of substitution ciphers," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9490, pp. 206–215, 2015.

[26] S. William and W. Stallings, "Cryptography and Network Security, 4/E," *Pearson Education India*, 2006.

[27] S. Picek, M. Golub, and D. Jakobovic, "Evaluation of crossover operator performance in genetic algorithms with binary representation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6840, pp. 223–230, 2011.

[28] T. Bäck, D. Fogel, and Z. Michalewicz, "Handbook of evolutionary computation," *Release*, vol. 97, no. 1, p. B1, 1997.

[29] L. Chambers, *Practical Handbook of Genetic Algorithms*, CRC Press, 1998.

[30] J. Walker, Ent: A pseudorandom number sequence test program, Software and documentation available at http://www.fourmilab.ch/random/.

[31] C. Lamenca-Martinez, J. C. Hernandez-Castro, J. M. Estevez-Tapiador, and A. Ribagorda, "Lamar: A new pseudorandom number generator evolved by means of genetic programming," in *Parallel Problem Solving from Nature-PPSN IX*, pp. 850–859, Springer, 2006.

[32] G. Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, vol. 2, addition wesley, Reading, Massachusetts, Mass, USA.

[33] M. M. Meysenburg and J. A. Foster, "Randomness and ga performance, revisited," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1*, pp. 425–432, Morgan Kaufmann Publishers Inc, 1999.

[34] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Flannery, Numerical Recipes in C*, vol. 2, Cambridge University Press, 1982.

[35] G. Marsaglia, Yet another rng, Posted to the electronic billboard sci. stat. math, August 1.

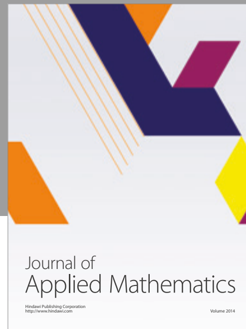
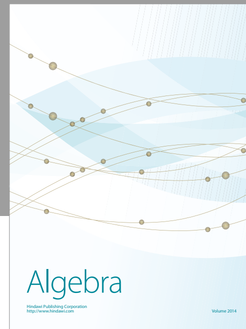
[36] M. Matsumoto and Y. Kurita, "Twisted GFSR Generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 2, no. 3, pp. 179–194, 1992.

[37] B. Schneier, "Applied cryptography," *Cover and Title Pages*, pp. 125–147, 1997.

[38] S. Tezuka and P. L'Ecuyer, "Efficient and Portable Combined Tausworthe Random Number Generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 1, no. 2, pp. 99–112, 1991.

[39] B. C. Johnson, "Radix-b extensions to some common empirical tests for pseudorandom number generators," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 4, pp. 261–273, 1996.

Advances in
Operations Research

Advances in
Decision Sciences

Journal of
Applied Mathematics

Algebra

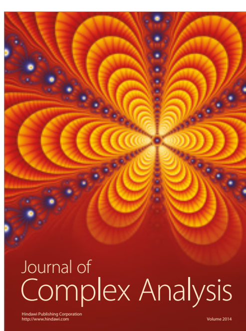
Journal of
Probability and Statistics

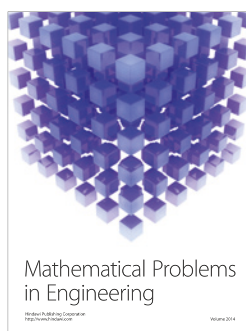
The Scientific
World Journal

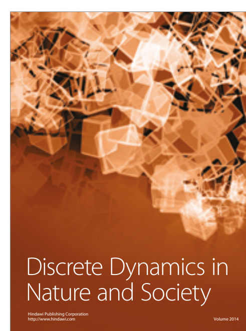
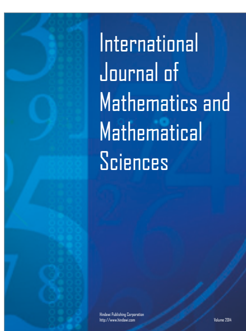
Hindawi
Submit your manuscripts at
https://www.hindawi.com

International Journal of
Differential Equations

International Journal of
Combinatorics

Advances in
Mathematical Physics

Journal of
Complex Analysis

Journal of
Mathematics

Mathematical Problems
in Engineering

Abstract and
Applied Analysis

Discrete Dynamics in
Nature and Society

International
Journal of
Mathematics and
Mathematical
Sciences

Journal of
Discrete Mathematics

Journal of
Function Spaces

International Journal of
Stochastic Analysis

Journal of
Optimization