Journal of King Saud University – Computer and Information Sciences 31 (2019) 92–104

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

κ-Lookback random-based text encryption technique

Muhammed J. Al-Muhammed, Raed Abu Zitar *

American University of Madaba, Faculty of Information Technology, Jordan

ARTICLE INFO

Article history:
Received 12 July 2017
Revised 18 September 2017
Accepted 6 October 2017
Available online 12 October 2017

Keywords:
Text encryption
Lookback text encryption
Random encryption operations
Security random generator

ABSTRACT

Although many encryption methods are available, there is always an ongoing need for more to resist the adversaries' ever-growing analytical skills and techniques. We propose in this paper an innovative method for text encryption. We devise a random number generation function that creates sequences of signed random numbers that depend on both plaintext and key. The random numbers support the functionality of four random operations: random mutation, random cyclic shifting, random permutation, and dirty symbol random insertion. These operations ensure the data security by steadily melting the statistical structure of plaintext and relationships to a key. The experiments with our prototype implementation showed that our method has high effectiveness (in terms of diffusion, confusion, avalanche) and high efficiency with respect to the computation demands.

1. Introduction

Securing information is a very important task. Good encryption methods must challenge the attackers' analytical capabilities by (1) eliminating the information in ciphertexts that may lead to predicting keys and (2) weakening the relation of the ciphertexts to both the keys and plaintexts. Consider for instance the plaintexts "love j**a**m" and "love j**i**m", which differ in the **boldfaced** symbol. Effective encryption methods must detect this tiny difference and reflect it by producing drastically different ciphertexts for them. Fig. 1 shows the plaintexts and their corresponding ciphertexts in unicode and Hex. As it could be seen, the encryption method responded to this tiny difference by producing largely different ciphertexts. In fact, it has magnified this tiny difference in such a way the large similarity between the plaintexts is highly vanished in their respective ciphertexts. Analytical techniques that look for specific patterns to predict keys are unlikely to succeed in such situations.

Many encryption algorithms have been devised (Preneel and De Canniére, 2006; Whiting et al., 1998; Rose, 1998; Sidney et al., 1998; Burnwick et al., 1999; Daemen et al., 2001; Biham et al., 2000; Nagaraj et al., 2013; Steef et al., 2015; Singh and Singh, 2015; Kamalakannan and Tamilselvan, 2015) and others (see Stalling, 2016). The advanced encryption standard (AES) stands out as one of the most effective encryption algorithms (The NIST test suite, 2016). This algorithm uses a secrete key and mathematical transformations that are based on lookup tables and predefined permutations to encrypt information. The complexity of the mathematical transformations makes AES strong and still unbreakable.

This paper proposes an innovative block encryption method. Our method defines an innovative random number generator and four operations to encrypt blocks of plaintext. The random generator produces sequences of signed random numbers using blocks of plaintext and a key. In particular, the generator looks back in plaintext k consecutive blocks and uses these k blocks in addition to a key to produce signed random number sequences. The random generator is very sensitive to plaintexts and keys. Change to a bit or more in plaintexts or keys causes very large variations to generated random sequences.

The signed random number sequences enable our method to control the functionality of the encryption operations. These operations use the random sequences to dynamically adjust their functionality and employ the adjusted functionality in encrypting blocks. For instance, the bit shift operation determines the amount of the shift and its direction (left or right) according to respectively the sign and value of the random numbers in a sequence. Furthermore, the random sequence allows our method to update the key after encrypting each block. The next block thus is encrypted using updated key.

* Corresponding author.
E-mail addresses: m.almuhammed@aum.edu.jo (M.J. Al-Muhammed), r.abuzitar@aum.edu.jo (R.A. Zitar).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2017.10.002
1319-1578/

Text	Ciphertext (Unicode)	Ciphertext (HEX)
love jam	àÏù› Í$	e0, cf, f9, 3e, 82, cd, ac, 24
love jim	mµËóÜ3ÃJ	6d, b5, cb, f3, dc, 33, c3, 4a

Fig. 1. An example of our encryption method output.

We delineate the steps of encrypting a block B_i as follows. The random generator looks back k consecutive blocks prior to B_i and uses them along with the key to generate a random sequence. The sequence adjusts the functionality of the encryption operations and, in a way to be made clear later, encrypt the block B_i. Thus, to encrypt B_i, our method not only considers B_i, but also k-previous blocks. This effectively makes encrypting B_i depends not only on B_i per se, but also on k-previous blocks and is affected by them.

The paper makes the following contributions. First, it offers an innovative random generator whose output strongly depends on plaintext and a key. This generator is augmented with effective noise detection and handling mechanisms that greatly magnify its sensitivity to variations of plaintext and keys. Second, it proposes encryption operations whose functionality is dynamically adjusted based on the output of the random generator. Third, it offers an effective mechanism for updating the encryption key.

We present our contributions as follows. Section 2 presents the random generator. Section 3 introduces our random encryption operations. Section 4 discusses our encryption method. We analyze the performance of our method in Section 5, conclude and give directions for future work in Section 6.

2. Random numbers generator

In this section we describe the fundamental components of our random generator. The generator requires an explicit mapping knowledge that correlate random numbers to both plaintext and key. This explicit knowledge is encoded in terms of static knowledge and dynamic knowledge. The static knowledge is a mapping mechanism called mesh (Section 2.1). The dynamic knowledge is captured through operations (Section 2.2). Section 2.3 illustrates how the generator uses the mesh, a key, and the operations to map symbols of plaintext and produce random numbers. Section 2.4 discussed how the noise that occur due to the key/plaintext changes is captured and embedded in the random generator. Section 2.5 presents the innovative properties of the generator that make it powerful for data encryption.

2.1. The mesh

The mesh provides a mechanism for mapping plaintext symbols and produce random numbers for these symbols. Fig. 2 shows an example of a mesh. The mesh is an $N \times N$ array with horizontal and vertical dimensions. Unicode symbols are listed in each dimension. The positions of the symbols in each dimension are indexed by integers 0, 1,. . .,n. Each cell in the mesh is a point (x, y), where x and y are respectively the vertical and horizontal indices of the cell.

The length of a move from point P_1 to P_2 is called a *distance*. The *horizontal distance* of a move from P_1 to P_2 is the **absolute** value of the difference between the horizontal indices of these two points. The *vertical distance* between P_1 and P_2 is the **absolute** value of the difference between their vertical indices. For instance, the horizontal distance between $R_2(1,6)$ and $P_3(1,4)$ is abs(6 – 4) = 2.

Since every move in the mesh starts from a point, we introduce the *move direction* with respect to a point. We designate a direction of a horizontal/vertical move with respect to a point P_1 by the flag "−" if the move is to a point with a lower index, and by "+" if the move is to a point with a higher index. Referring to Fig. 2, we designate the vertical move from R_1 to P_1 by "−" and the move from R_1 to P_2 by "+" because these moves are to respectively a lower/higher vertical indices in the mesh.

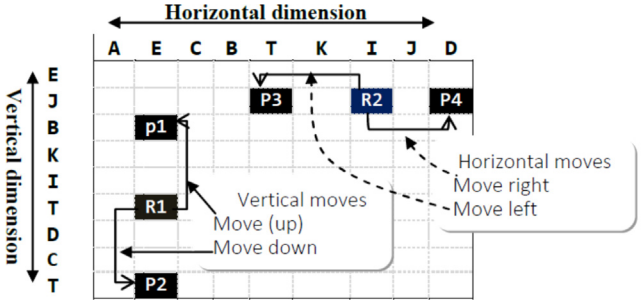

Fig. 2. An Example of a mesh.

We show here in a way to be made precise later how to generate signed random numbers using the mesh. We map a symbol by beginning from a starting point and move along one of the mesh's dimensions to the corresponding index of this symbol in that dimension.[1] We call the dimension that we move along a *mapping dimension*. The distance of the move along the dimension and its direction with respect to the starting point is compiled as a **signed random number**.

2.2. Operations

The operations control the state of the random generator. Three operations are used: (1) ordering operation, (2) mapping operation, and (3) key update operation.

2.2.1. Ordering operation

Ordering operation re-arranges the symbols of a mesh's dimension. It controls the state of the random number generator and its output. In fact, each different ordering of the mesh's symbols results in different mapping; causing the generator to produce a different sequence of random numbers even for the same text. To use the random generator in cryptography, we bind the generator's state change with the key. A change to the key modifies the functionality of the generator causing the resulting random numbers to vary.

We use the ordering operation *order*(*key*) whose functionality is defined as follows. Let n be the number of symbols in each dimension. The operation splits the symbols of each dimension into blocks of size ℓ symbols (in our implementation $\ell = 16$). These blocks are populated in the columns of an array of $\ell \times \jmath$, where $\jmath$ is the number of blocks. The rows are then left shifted. The amount of the shift depends on the key. The amount of the shift is a random number obtained using the procedure in Fig. 3.

Initially the procedure expands the key to 32 symbols (if shorter) using the same procedure in *AES* (Stalling, 2016) and use it as a seed-steps (1) and (2). The amounts of shifts are then generated using the steps (3)–(6). As the figure shows, the symbols of the seed are summed by multiplying the integer value of each seed's symbol k_i by its position in the seed (step 3). The sum is then circularly left shifted n positions to yield Sh (n is value of the first two symbols in the seed) in step 4 and Sh is XORed with the sum in step 5 to yield Ls. Since the number of symbols in each row is $\jmath$,Ls is adjusted to Ls module $\jmath$.

[1] A starting point is a point from which the random number generator starts. Any point whether in or out of the mesh can be used as starting point.

(1) *Expand the key*
(2) seed = key
Repeat
(3) $sum = \sum_{i=1}^{|seed|} i \times k_i$
(4) $Sh = LShift(sum,\ n)$
(5) $Ls = (sum \bigoplus Sh)\ \%\ \jmath$
(6) $seed = concatenate(seed, sum)$
Until *condition*

Fig. 3. Steps for generating row shift values. |*seed*| denotes the seed's size.

The seed is updated in step (6) by concatenating the seed with the current sum. The seed hence grows after each iteration. If, at any iteration, step (3) results in overflow in the sum, the procedure reduces the seed to 32 symbols by using the middle 32 unicode symbols as a new value for the seed. Steps (3) through (6) repeat until the condition no longer holds. (The condition determines the desired number of values to be generated.)

We illustrate the symbols' ordering using an example. Suppose a mesh's dimension has the symbols "*ABCDEFGHIJKLMNOPQRST*" and the key is "*A6Bf*". Fig. 4 shows the symbol arrangement process. The symbols are split into blocks of size, say, four. These blocks are placed in the array's columns as shown the figure. The key is used by the procedure in Fig. 3 to generate four numbers to shift the four rows. The ordering operation then uses these numbers to cyclically left shift each row a number of positions equals to the corresponding number. The first row is thus left shifted four positions, the second three positions, and so on. The operation outputs the new symbol's order by reading the symbols column-wise to obtain "*QNGHARKLEBOPIFSTMJCD*".

2.2.2. Mapping operation

The mapping operation defines the functionality for mapping plaintext's symbols to the mesh and generating signed random numbers. The operation selects, based on the key, one of the mesh's dimensions as the *mapping dimension*. It moves then from current starting point to the corresponding index of the symbol along the chosen mapping dimension and calculates the distance of the move. The direction of the move is translated as "−" or "+" according to whether the move to a lower or higher index.

We introduce the mapping operation *KeyBasedMapping*(*key*) that performs three actions: key-based dimension selection, distance calculation, and direction determination. The latter two actions are obvious and do not depend on the key. The former action (dimension selection) depends on the key for many reasons. First, the key should guide the switching between the two dimensions so that we can replicate the action. Second, the key in cryptography is ideally a random sequence of symbols; the switches between the mesh dimensions and the moves therefore follow this randomness.

We can correlate the dimension selection to the key in many ways. For instance, one can do this by considering the numerical value of the key' symbols: if the value of the current key's symbol is odd select the vertical dimension; the horizontal dimension otherwise. Another way is to find the median of the key's symbols and select, say, the vertical dimension if the current key digit is less than the median value and horizontal dimension otherwise.

2.2.3. Key update operation

The key is a secret sequence of n unicode symbols $k_1 k_2 \ldots k_n$, which will be used to generate signed random numbers for the random operations (discusses next). To weaken the relationship between a key and ciphertext, the key is updated before encrypting each block. We use a sequence of signed random numbers $r_1 r_2 \ldots r_n$ to update the key. We add each signed random number r_i to the value of its corresponding key symbol k_i. The outcome of this is the updated key. For instance, consider the key "*AB*" and the sequence of random numbers "+12-2". Adding the symbols of the key to their respective signed random numbers [*A(65) + 12 = 77 (M), B(66)-2 = 64 (@)*] yields the updated key "*M@*".

Although we linearly add the key symbols to the random numbers, the outcome is not. In fact, adding key's symbol to a signed random number grows and shrinks according to the sign of the random number. Since the distribution of the random number sequence's signs is random, this grow or shrink in the addition outcome is random.

Fig. 5 shows the subsequent updates of the key "97A12ag678c v6435". The first row shows the original key. The rest of the rows show the updated versions of the key. As can be seen, each new version is largely different from the previous ones.

2.3. Random number generation process

The symbol ordering, the starting point, and the key define the random generator state. Given a state, our random generator produces a sequence of signed random numbers as follows. First, the generator re-arranges the symbols of each dimension using the operation *order*(*key*). Second, the generator reads a symbol t_i from plaintext and a symbol k_i from the key. It then selects the mapping dimension based on the key symbol k_i and moves along this mapping dimension to the index of the symbol t_i. The generator calculates the distance of the move from the current starting point to the symbol t_i by subtracting their indices on the mapping dimension. The move direction—within the mapping dimension—with respect to the current starting point is determined based on whether we move to a lower index ("−") or a higher index ("+"). The distance of the move d in addition to the direction of this move "±" are compiled into a signed random number "±d".

We illustrate the random generation process using an example. Consider the plaintext "*To be or Not to be.*", the key "*919981237659112348*", and a starting point (9, 8). Table 1 shows

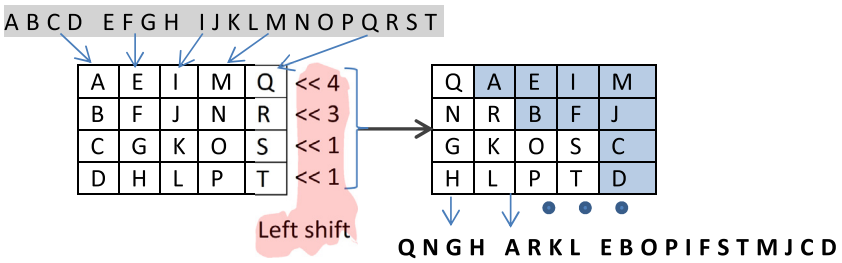

Fig. 4. The symbol re-arrangement process.

```
-----------------------------------
|Original key :  97A12ag678cv6435  |
|First update :  1#R?_DkH!$\?<6/-  |
|Second update:  K3LW_K]07.1~;I_3  |
|Third updat  :  :G>=_fjD_1tnN4+1  |
-----------------------------------
```

Fig. 5. The original key and the result of three updates of the key.

Table 1
The generated random numbers for the text *To be or Not to be.*

Plain Text:	To be or Not to be.
Key:	919981237659112348
Random Numbers	−6-1 + 5-3-0 + 3-1-6 + 6-7-5 + 2 + 2-1 + 7 + 2-1 + 2-7

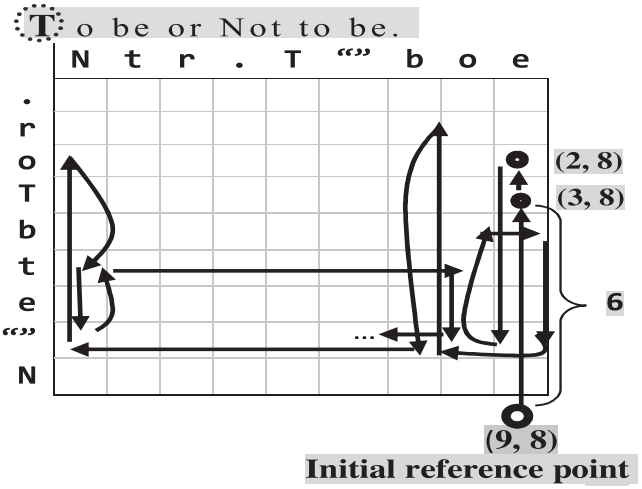

Fig. 6. Mesh's configuration during mapping the symbols.

the generated sequence of signed random numbers for the plaintext using the key and starting point. Fig. 6 shows the moves within the mesh during the symbol mapping operation (partial).

Initially, the random number generator re-arranges the mesh's symbols using the operation *order*(*key*), yielding the symbol organization in Fig. 6. The random number generator reads the symbol "T" from the plaintext and the leftmost key's digit "9". Selecting the mapping dimension depends on the value of the current key symbol and the functionality of the operation *KeyBasedMapping*(*key*). Suppose (for simplicity) that the *KeyBasedMapping*(*key*) "*select the vertical dimension if the value of the key's digit is odd; the horizontal dimension otherwise*." Because the value of the current key's digit is 9, which is odd, this operation selects the vertical dimension as the mapping dimension. The distance of the move from the initial starting point (9, 8) to the symbol to be mapped along the vertical dimension is 6. In addition, because the index of "T" within the mapping dimension is at a lower index than that of the initial starting point, the generator designates this by the flag "−". The generator produces the signed random number "−6" for "T". The generator updates the starting point to (3, 8) and uses it to generate a random number for "o". The value of the current key digit is 1 (odd), the mapping dimension is therefore the vertical dimension. The distance of the move from the point (3, 8) to "o" is 1. The sign of the random number is "−" because we moved to a lower index in the mapping dimension. Hence, the signed random number for the symbol "o" is "−1".

Continuing likewise, the generator moves in the mesh as shown in Fig. 6, producing a signed random number for each symbol and yielding the sequence of signed random numbers listed in Table 1.

2.4. The key/plaintext noise

Ideally, a change to a key or plaintext, which we call *noise*, must cause large changes to the resulting sequence of random numbers. This is an essential property for the effectiveness of the generator in the filed of cryptography. Relying on the mapping alone allows the changes to the text or the key to only affect the sequence of random numbers from the position of the change and on. To propagate the change to the whole sequence not only from the point where the change occurs, we augment our random generator with a noise handling technique.

The noise handling technique captures and then reflects the noise in the resulting sequence random numbers. There are many ways to do so, ranging from the most intuitive one by adding the amount of noise to each random number to model the noises as random pulses. We prefer the latter, however. We use random number generators to model the noises as random pulses because these generators not only enable capturing the intuitive meaning of the noise (random effect), but also enable distributing the noise's random effect over all the generated random numbers sequence.

Although many random number generators (L'Ecuyer, 2012; Shub et al., 1986), we used the register shifting random generators (called Xorshifting) to model the noise (Marsaglia, 2003; L'Ecuyer, 2005) because they are fast, easy to use, and requires less storage. The register shifting generator produces sequences of random numbers by applying shift and XOR operations to a seed (an integer). Fig. 7 shows the register shifting and the XORing for the seed x along with the amount of the shifts a, b, and c. Changes to the seed yield changes to the random numbers sequence (i.e. causing a noise). We exploit this idea in our noise handling by computing the amount of the change in key/plaintext symbols and use it as seed for the random generator. We generally sum the symbols of the key and the symbols of the text by multiplying each symbol value by its position in the key (or the text). The symbol value for digits is the value of the digit itself while the value of other symbols is the unicode index of these symbols. We add the two sums to yield a seed. For instance, for the symbols "5A8" the sum is "5*1 + 65*2 + 8*3 = 159". Note that 65 is the unicode's index of the symbol "A".

Given this, we describe our mechanism to propagate the noise as follows. We compute the amount of noise and use it as a seed for the Xorshifting generator. We then use the Xorshifting generator to produce a number of random pulses equal to the number of random numbers in the initial sequence. Finally, we add each noise to the corresponding random number regardless of the sign, yielding a new sequence of random numbers. Clearly, the new sequence is greatly correlated to changes of the key or text.

To illustrate, consider the key "*6739*" and the text "*ABCDE*". To generate noise–augmented random numbers for this text, we

$$\begin{aligned} x &= x \oplus (x \prec\prec a) \\ x &= x \oplus (x \succ\succ b) \\ x &= x \oplus (x \prec\prec c)) \end{aligned}$$

Fig. 7. Xorshifting Random number generator equations. $\prec\prec$ represents left shift, $\succ\succ$ represents right shift, and $\oplus$ represents Xor.

map the text "ABCDE" to the mesh and obtain the initial random sequence "*+18-7-3 + 12 + 24*". We then sum the symbols of the key (6*1 + 7*2 + 3*3 + 9*4) and the symbols of the text (65*1 + 66*2 + . . . + 69*5) to yield respectively 65 and 1015. We add the two sums (65 + 1015) to yield 1080, which is our seed. We pass the seed (1080) to the random generator, which returns the random pulses 27, 7, 4, 19, 20. We finally, add these random pulses to the initial random sequence to obtain noise–augmented random numbers "*+45-14-7 + 31 + 44*".

To further discuss the effectiveness of the noise handling mechanism, let suppose that the text "*ABCDE*" changed to "*ABC***H***E*". Our generator maps this text to the mesh and generates the sequence of initial random numbers "*+18-7-3***+22 + 12**". Observe that the random sequence is affected from position of the change and on. The sum of the modified text is 1031, which will be added to the sum of the key (65) to yield 1096. This value (1096) is passed to the noise generator, which returns the random pulses 7, 3, 13, 6,16. Adding these random pulses to the initial sequence yields the noise-augmented random sequence "+25-10-16 + 28 + 28". By a simple comparison between the sequences before and after the change, it is clear that the noise resulted from changing one symbol in the text caused a big change to the resulting random sequence—which exactly what we want.

2.5. Random generator properties

The output of our proposed random number generator depends on the state of the mesh and the key. The state of the mesh is controlled by the order of the symbols on the mesh dimensions and this order depends also on the key. In addition, the generated random numbers depend on the plaintext, allowing for relating the random numbers to the plaintext. As a result, one can generate different random numbers by either changing the key, changing the state of the mesh, plaintext, or of course all. Table 2 shows the impact of changing a key and the order of the symbols on the resulting random numbers. Referring to the table, the first 3 sequences of random numbers are different as a result of only changing the key and the second three sequences are also different as a result of changing the order of symbols, but using the same key. Table 3 shows the effect of making tiny changes to the key or plaintext.

Our proposed random number generator differs from other random generators. These generators either depend on some seed—basically an integer—to generate a sequence of pseudo-random numbers or physical generators whose sequence of random numbers depends on some physical noise. Both types of generators are unsuitable for encryption either because knowing the seed would certainly allow to regenerate the same random numbers (mathematical generators) or because it is not possible (or is not easy at best) to replicate these random numbers. Furthermore, these generators do not effectively relate the generated numbers to the plain text—our generator does.

In contrast to these generators, our generator depends on other information to generate random numbers in addition to a key. This makes it perfectly suitable for cryptography. First, the generated numbers are related to the key and plaintext. Second, knowing the key is far from sufficient to allow regenerating the random numbers because the random numbers depends also on the mesh state and plaintext. Third, the random numbers that are generated using a key K, a mesh state S, and a plain text T can be replicated using these three pieces of information. Furthermore, the random numbers are very sensitive to noises caused by changes to key/plaintext.

Our random generator guarantees high confusion and diffusion. As discussed above, any change to the text results in different random pulses (noises) and different mapping to the mesh. This logically causes large changes to the generated random numbers regardless whether the key change or not (high confusion). Likewise, any change to the key results in different random pulses (noises) and mapping to the mesh, leading to different random numbers. This means the relationship to the key is complex and involved one (high confusion). In addition the noise due to text changes spreads over all the generated random numbers (high diffusion).

Table 2
The effect of changing the key or order on the resulting random numbers.

Changing in	Resulting Random Numbers
KEY	−6-1 + 14-3-0 + 4-1-6 + 15-7-5 + 12 + 10-2 + 7 + 3-1 + 2-7 −3-4-4 + 11-2-1 + 1 + 5-4-3-1 + 9-2 + 11 + 3-2-2 + 1-3 −4 + 2+6-3 + 4 + 12-1-6 + 15-7-5 + 11 + 3-2 + 7 + 4-1 + 2-7
Symbol Order	+20-10 + 3 + 4-5-2-2 + 13-3 + 16-8 + 3 + 11-2-11-6-7 + 7 + 10 +3 + 4-5-1 + 14-0-4 + 10 + 9+5-4 + 4+4 + 3-8 + 8-8 + 9+7 −1-7 + 22-11-4 + 12-3-2 + 11 + 11-13 + 10 + 20-12-11 + 13 + 5+8-4

Table 3
The effect of the noise that results from tiny changes to the key and text. The changed symbols are underlined.

Text	Oxygen necessary
9836908761534798	−16-8 + 34 + 23-11 + 25 + 32 + 27 + 5-7-10 + 15-19-29 + 9-16
9836908761534799	−19-9 + 19 + 32-26 + 30 + 8 + 33 + 3-23-27 + 39-25-32 + 34-32
98369<u>2</u>8761534798	−20-14 + 37 + 10-18 + 25 + 29 + 28 + 23-16-30 + 24-20-24 + 10-10
<u>7</u>836908761534798	−22-11 + 24 + 16-22 + 8 + 29 + 34 + 12-27-2 + 29-26-18 + 23-4
Key	9836908761534798
Oxygen necessary	−16-8 + 34 + 23-11 + 25 + 32 + 27 + 5-7-10 + 15-19-29 + 9-16
Oxyge<u>i</u> necessary	−8-21-16 + 18-9 + 27-33 + 42-9 + 40-26 + 14 + 27 + 40-31 + 24
Oxygen necessar<u>y</u>	−6-27-30 + 17-6 + 16-17 + 33-16 + 17-13 + 37 + 35-12 + 25-17
<u>p</u>xygen necessar<u>y</u>	−17-30-15 + 17-29 + 6-39 + 35-9 + 17-22 + 34 + 14-11 + 13-21
<u>pm</u>ygen necessary	−33-14-20 + 24-14 + 19-39 + 34-28 + 17-9 + 23 + 41-10 + 26-10

3. Random-based operations

We describe here the operations that our encryption method requires to function. First, the method requires random mutation for mutating symbols of plaintext (Section 3.1). Second, it requires random shift to diffuse the symbols and mix them (Section 3.2). Third, it requires further scattering of the plain text symbols (Section 3.3). Finally, it requires a way for melting the boundaries between the blocks of ciphertext (Section 3.4).

3.1. Random mutation operation

This operation takes two sequences of signed random numbers $r_1^1, r_2^1, \ldots, r_t^1, r_1^2, r_2^2, \ldots, r_t^2$ and a block $b_1, b_2, \ldots, b_t$ as an input and returns a block of randomly mutated symbols $m_1, m_2, \ldots, m_t$. The operation performs double non-linear random transformations to randomly mutate the block's symbols. The random transformation makes use of two one-way arrays, which we denote $SBOX_1$ and $SBOX_2$. These two arrays are filled (at run time) with unicode symbols and then independently shuffled using a key–based behavior identical to that used in mesh reordering (Fig. 3). The random

Table 4
An example of $SBOX_1$ and $SBOX_2$ (partial).

$SBOX_1$	'	ó	%	#	B	>	~	Ê	ü	ˆ	v	¬	e	V	\	÷
$SBOX_2$	3	H	%	Z	ý	#	{	©	ö	Å	A	Ì	Ý	3/4	*	÷

transformation uses two random numbers r_i^1 and r_i^2 to respectively index $SBOX_1$ and $SBOX_2$ and retrieve two unicode symbols ψ_i and η_i. These two symbols ψ_i and η_i are then XORed with the block's symbol b_i to yield m_i. That is, $m_i = b_i \oplus \psi_i \oplus \eta_i$.[2]

We illustrate our random mutation operation using a simple example. Consider the plain text "Encrypt me". Let "−10 + 7 + 12-5-6 + 8-1 + 13-4-15" and "+4 + 11-5-6-3 + 9–2 + 14 + 3-8" be two sequences of signed random numbers. Table 4 shows a partial example of $SBOX_1$ and $SBOX_2$.

The mutation operation mutates the given text symbols using the two sequences of random numbers as follows. It reads the first two signed random numbers "−10" and "+4" from the sequences and the first plain text symbol "E". The two random numbers "10" (without the sign) and "4" are used to respectively index $SBOX_1$ and $SBOX_2$ and retrieve the unicode symbols "v" and "ý". The retrieved symbols are XORed with the plaintext symbol "E" to yield the symbol Î, which is the mutation of "E". The next two random numbers and plaintext to be read are respectively "+7", "+11", and "n". These two numbers index $SBOX_1$ and $SBOX_2$ and respectively retrieve the two symbols Ê and Ì. The mutation operation XORes these two retrieved symbols with the plain text symbol "n" to yield the mutated symbol "h". Proceeding likewise, the mutation operations randomly mutates all the symbols of the plaintext "Encrypt me" and outputs the mutated text "Îh%7]Iø\ud".

The mutated text is highly correlated to changes of the key and the text. That is because these key/text changes alter the generator state and necessarily lead to different mappings and different random pulses (noises). The generator responds to the change in its state by making drastic modifications to the generated random numbers. Different random numbers retrieve different unicode symbols and therefore result in different mutation outcomes.

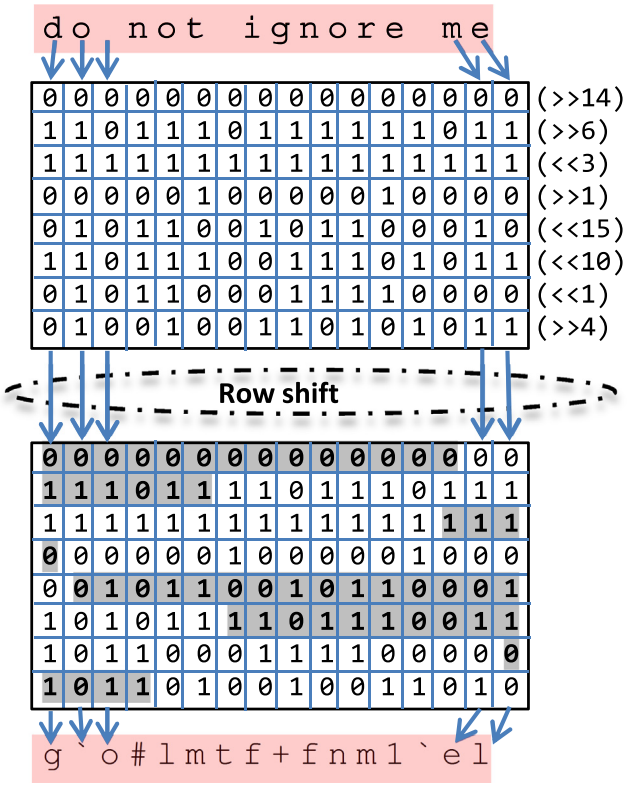

Fig. 8. Row shifting. Each row is shifted randomly x positions either to left $\ll x$ or to right $\gg x$.

3.2. Random cyclic shift operation

This operation performs bitwise shifts to a block of symbols. We obtain the bit representation for each symbol in the block. These bits are placed in $m \times n$ array, where m the number of bits that represents a symbol and n is the number of symbols in the block. The number of bits m depends on the encoding system. Thus, m could be 8 or 16 bits depending on the used encoding. Each symbol occupies one column of the array, where the first symbol occupies the leftmost column.

The operation performs a random cyclic shift for each row in the array. The amount of the shift for each row and the direction of this shift (left or right) is fully determined by a sequence of signed random numbers r_1, r_2, etc. The row i is shifted a number of positions equal to the absolute value of r_i. The direction of the shift depends on the sign of r_i. Generally, the row i is left shifted if the sign of r_i is negative and is right shifted if the sign of r_i is positive. Observe, we try to capture the intuitive meaning of the sign: positive is to the right of the zero (on the number line) and negative is to the left.

Fig. 8 shows an example of row shift, where each symbol is represented using 8 bits. Referring to the figure, the rows are shifted by different amounts and to different directions. For instance, row 2 is right shifted by 6 while row 3 is left shifted by 3.

It is possible to reverse the shift operation and obtain the original block. We cyclically shift the rows of the array as before. The only difference is that the direction of the shift must be reversed. That is, we left shift row i if the sign of the random number r_i is positive ("+") and right shift the row if the sign is negative ("−").

The operation randomly mixes the symbols of a block at the bit level. This fine-grained mixing of the block symbols' bits brings a large confusion to the encryption. It specifically further weakens the relations between the input block and its respective ciphered one because the arrangement of bits in the input block is significantly (and randomly) modified in the output block.

3.3. Random permutation operation

Random permutation involves two operations: Random Crossover and Random Repositioning. These two operations work synergistically to diffuse block's symbols over large number of other block symbols.

3.3.1. Random crossover

Crossover operation produces new symbols (or breeds) by exchanging a random number of bits between pairs of randomly selected symbols $b_1 b_2 \ldots b_n$ of a block. A sequence $r_1 r_2 \ldots r_n$ of signed random numbers determines both the pairs of symbols to exchange some of their bits and the number of bits to exchange. For each two consecutive random numbers $r_i r_{i+1}$ (i = 1,..., n-1), the symbol at the position r_i % n exchanges the first or the last of its m bits with the first or the last m bits of the symbol at the index r_{i+1} % n. The number of bits m is $MAX(r_i \% s, r_{i+1} \% s)$, where s is the number of bits that represent a symbol.

[2] We call the mutation, random mutation, because the unicode symbols are randomly indexed.

Since the distribution of the signs in the random sequence is random, it is logical to base whether to exchange the first or the last m bits of a symbol b_i on the signs of the sequences $r_i r_{i+1}$ (i = 1,..., n-1). The first m bits of the symbol b_i that is indexed by r_i are exchanged if the sign of r_i is negative; the last m bits are exchanged otherwise. Similarly, the first or the last m bits of the symbol b_j are exchanged if the random number r_{i+1} is respectively negative or positive.

To illustrate, consider a block of three symbols "the" whose binary representation is "01110100 01101000 01100101" and a random sequence "−1 + 3 + 2". The subsequence "−1 + 3" indexes the symbols t and e whose binary representations are respectively "01110100" and "01100101". The maximum number in the subsequence is 3. Since the first random number in the subsequence (−1) is negative, the first three bits "011" of t representation are exchanged. Likewise, since the second random number in the subsequence is positive (+3), the last three bits "101" of e representation are exchanged. This gives the new breads "**101**10100 01101000 01100**011**". The subsequence "+3 + 2" indexes—in the new breed—respectively "01101000" and "01100011". The last three bits of these two binary representations are exchanged. This gives the new bread "01101**011**" and "01100**000**". As a result, the crossover between the symbols of "the" gives the new bread "**101**10100 01101**011** 01100**000**" (or in symbols 'k').

The crossover operation is reversible only if the random number sequence $r_1 r_2 \ldots r_n$ is known. To reverse and obtain the original block, we perform the same steps but backwards. That is, we start from the last two random numbers $r_{n-1} r_n$ of the sequence and let the symbols at these indices exchange bits as describe above. We continue likewise until the symbols at first two random numbers $r_1 r_2$ exchange bits.

The crossover operation mixes the input block's symbols at a coarser-grained level than the bit shifting operation. Both the consecutive bits to be exchanged and their number are randomly selected. This random manipulation, therefore, to the symbols of the input block makes drastic changes to the structure of its bits. Consequently, the structure of the output block greatly differs from that of the input block.

3.3.2. Random repositioning

This operation randomly reorders the symbols of a block. It does so using a sequence of signed random numbers $r_1, r_2, \ldots, r_n$. Each two consecutive random numbers r_i, r_{i+1} are used to swap two symbols of the block whose positions equal to these random numbers. More specifically, consider the block symbols $s_1\ s_2\ \ldots s_p\ \ldots s_t\ \ldots s_n$ and the random numbers $r_1, r_2, \ldots, r_n$. For each two consecutive random numbers r_i, r_{i+1} (i = 1, 2, ...n-1), the process swaps the two symbols, say s_p and s_t whose positions in the block equal to r_i, r_{i+1}. The result is the new sequence $s_1\ s_2 \ldots s_t\ \ldots s_p\ \ldots s_n$.

To illustrate, consider the block "*ABCDEFG*" and the signed random numbers "−1 + 3 + 7-4 + 2+6-5". To reorder these symbols using the given random numbers, we start with the first two consecutive numbers "−1 + 3". The operation swaps the symbols at the positions "1" and "3", namely the symbols A and C, yielding the new permutated sequence "*CBADEFG*". Next the operation takes the second and third random numbers "+3 + 7" and swaps the symbols at these two positions (*A* and *G*), yielding "*CBGDEFA*". Continuing likewise, we obtain the randomly permutated block "*CFGBAED*".

Obtaining the original block from the permutated one can be performed using the same sequence of random numbers $r_1, r_2, \ldots, r_n$. The order in which the permutation is carried out is different, however. The permutation operation swaps the block symbols backwards starting from the last two random numbers r_{n-1}, r_n. Therefore, it swaps the symbols at the positions r_n and r_{n-1}, then the symbols at the positions r_{n-2}, r_{n-1}, and so on until the symbols at r_1, r_2.

3.4. Dirty symbol insertion operation

This operation inserts dirty symbols in randomly chosen blocks. We base our random choice of the blocks on the sequence of random numbers. The idea is to consider the sequence of random numbers $r_1, r_2, \ldots, r_n$ and count the number of plus and negative signs. If the number of pluses equals to number of minuses, the operation inserts a dirty symbol at the position $r_i \% (m-1)$, where r_i is the first non-zero random number in the sequence and m is the block size. The dirty symbols is a unicode symbol chosen from $SBOX_1$ at the index r_1.

Inserting dirty symbol changes the structure of blocks. It modifies the boundaries between the blocks causing them to overlap. If the decryption process fails to correctly recognize and remove the inserted dirty symbol from a block, the decryption of all the blocks that follow the first unrecognized dirty byte fails. That is because the structure of the next blocks in the ciphertext does not match the structure of the original plaintext blocks (due to the extra symbol). The decryption of this block thus yields a block with different plaintext than that of the original one. Since the previously decrypted block will be an input for the random generator, the random generator responds to this wrong input by producing a wrong sequence of random numbers. Consequently, the random operations whose functionality depends on the random numbers fail to correctly recover the current block. This error propagates to all of the next blocks because their correct decryption depends on the correct decryption of their predecessors.

4. The encryption method

We discuss in this section our cipher. Section 4.3 discusses how to encrypt plaintext using the random operations. Section 4.4 discusses the decryption process. But before we discuss the technical details of our encryption method, we discuss how we select random sequences.

4.1. Random blocks generation

Initially, when our method starts encrypting the first plaintext block, there is no yet predecessor plaintext blocks to use in the lookback technique. To enable the lookback, the encryption method elects k blocks of 16 symbols each from the two mesh's dimensions. The blocks' election is based on the key. We use the method in Fig. 3 to generate indices for accessing the two dimensions of the mesh and combine the respective symbols to create the k blocks.

Given a set of indices $i_1, i_2, \ldots, i_u$ ($u = 16k$), our method selects unicode symbols from the mesh by alternating between the two dimensions starting from the horizontal one. That is, the method selects the unicode symbol Π_{i_1} at index i_1 from the horizontal dimension, the symbol Π_{i_2} at index i_2 from the vertical dimension, and so on till Π_{i_u}.

The encryption method uses the created k blocks to generate sequences of random numbers for the encryption process. But as we see next, as the encryption process progresses, these initial k blocks will be thrown away one by one and replaced by blocks from the plaintext to be encrypted.

4.2. Random sequence selection

Since our algorithm looks back k blocks each of size 16, the random generator produces the sequences of random numbers

$\Omega_1, \Omega_2, \ldots, \Omega_k$, where each sequence Ω_i consists of 16 signed random numbers. In addition, our method consists of seven random-based operations each requires one sequence for its functionality. Therefore, the encryption process selects only seven sequences out of k sequences Ω_i ($i = 1, 2, \ldots, k$). The sequence selection mechanism is randomly based. Let Λ be a random variable whose possible values are the sequences Ω_i ($i = 1, 2, \ldots, k$). We consider every sequence Ω_i has the same probability $p_i = 1/k$ to be selected. To select a value Ω_j for Λ, we use the computer built-in random generator seeded with the encryption key to generated random numbers ψ_i in [0, 1]. Based on outcome of the random generator, we select $\Lambda = \Omega_i$ if $\psi_i < \sum_{j=1}^{i} p_j$.

For instance, if we assume that our method looks back $\kappa = 8$ blocks, the generated sequence of random numbers is $\Omega_1, \Omega_2, \ldots, \Omega_8$. Let us also suppose that the probability of selecting any of Ω_i is $p_i = 1/8 = 0.125$. If the generated random number is $\psi = 0.36$, the selected sequence is $\Lambda = \Omega_3$.

4.3. Encryption process

This section discusses how our encryption process uses the random number generator and the random operations to encrypt plaintext. The input to the process is a 16-symbol key K and plaintext T and the output is a ciphertext C. It splits the plaintext T into blocks $B_0, B_1, \ldots, B_n$. Each block B_i is 16 symbols except probably the last one, which may have fewer. Fig. 9 illustrates the encryption process.

The encryption process has two parts. The first part encrypts the block B_0 using a sequence of signed random numbers that is generated using the key and k random blocks. In the second part, our method starts using the previous blocks of the plaintext in the random number generation process. That is, the second part produces a ciphertext C_i for a block B_i using a sequence of random numbers that is generated using the key and the previous blocks $B_0, B_1, \ldots, B_{i-1}$ along with random blocks if needed to complete the input of the random generator to k blocks. Note after encrypting the first k blocks of plaintext, the method no longer needs the random blocks and the random generator uses only the previous k blocks of plaintext.

In the first part, the random number generator maps the symbols of the k blocks to the mesh and generates the sequences $\Omega_1, \Omega_2, \ldots, \Omega_k$ of signed random numbers. (Each subsequence Ω_i consists of 16 signed random numbers.) Let $\Omega_1, \Omega_2, \Omega_3, \Omega_4, \Omega_5, \Omega_6, \Omega_7$ be the seven sequences randomly selected from Ω_i ($i = 1, 2, \ldots, k$). Referring to Fig. 9, the encryption process passes the sequences as an input for the random operations. It then executes the random operations in the specified order to encrypt the block B_0 and outputs the ciphertext C_0. The process first executes the random mutation operation. The random mutation operation uses the first two sequences Ω_1 and Ω_2 to mutate B_0 symbols $b_1, b_2, \ldots, b_{16}$ as discussed in Section 3.1. The output is the mutated block $m_1, m_2, \ldots, m_{16}$.

The mutated block $m_1, m_2, \ldots, m_{16}$ is passed to the random shift operation. Since the number of rows to be shifted is 8, the shift operation uses the first 8 random numbers of the sequence Ω_3 to shift the bits of the block as described in Section 3.2. The output of the random shift operation is passed to the random permutation operation.

The permutation operation uses the sequences Ω_4 and Ω_5 to randomly cross over and scatter (reposition) the symbols of the shifted-mutated block. Specifically, the crossover operation uses the sequence Ω_4 to cross over the input block. The repositioning operation uses the sequence Ω_5 to scatter the symbols of the input block. The output of the permutation operation is passed to the dirty-symbol insertion operation. This operation uses the sequence Ω_6 to insert a unicode symbol in a block as described in Section 3.4. The output is the ciphertext C_0 for the block B_0.

For the remaining plaintext blocks $B_1, B_2, \ldots$(if any), the encryption process follows the same previous steps, but with two fundamental modifications. Fig. 9 shows these modifications. First, the encryption process updates key before encrypting any new block B_i ($i = 1, 2, \ldots$). The key is updated using the procedure in Section 2.2.3 and the sequence Ω_7. Second, the input to random generator changes. As Fig. 9 shows, the generator starts receiving plaintext's blocks $B_0, B_1, \ldots, B_{i-1}$ and random blocks R_i if needed

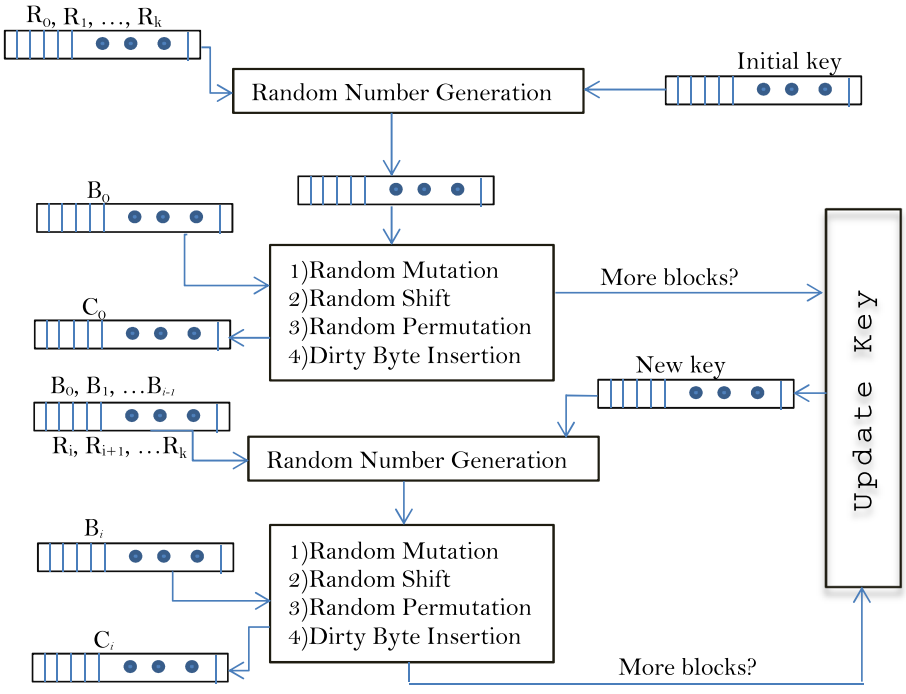

Fig. 9. The encryption stages. B_i, C_i, and R_i are respectively the block i of plaintext, the block i of ciphertext, and the random blocks.

Plain Text (301 symbols including white spaces, taken from Wikipedia)	KEYS Versions. First one the original key	Sample of random numbers. Each sequence for one block
Banksia canei (mountain banksia) is a shrub of the subalpine areas of the Great Dividing Range between Melbourne and Canberra in southeastern Australia. First collected on 27 November 1962, it superficially resembles B. marginata, but is more closely related to another subalpine species, B. saxicola.	p7A12ag798cv6432	-11-16-28-13-9+34-31-19+20+28+5-35+18-25-17-23
	e'%$)?H$MThSH_"_	-6-18+32+30-30-35-11+19-10-23+38-16-32+10+13-27
	__EB_`=7C=?C(%/	**+16-18-15+26-22-10+26+17-8+11+32-30+4-27-37+36**
	o_6\?VWH;H®%,	**-35-31+28+9+27-10+17+30-33+16-13+26+17-24-32-11**
	L?Re_Lhf_X¡?=??_	-6-27-29+22-7-29+32+34-34-33+8-1-12+24+35-31
	DÉhO)d Q<1?PU_Û?	**-8-27+22-22+25+24+23-21+34-39-17+17+24+19-15-26**
	>®Ke"G?s__?OI?þà	-9-31+29+19-7+6-14-16+21+16-34+15+36+23-4+14
	M?HB+$½k/?¯k;__À	-25-24+36-11+3+21-33+19-9+9-20+20-27+33+12+26
	D{eU$*¯[D_?z___Î	-11-12+29-26-12-13+25-24-34+8+16-6-15-15+20+19
	+c?J'??n;_y?D@#è	-28-3-16-14+30-21+26+14-17+5-29+31+27-3-31+26
	Ð)?Pßå%$A[Ì_=Wõ	-16+11+8-18+11-27+27-26+29-25-25+26+11+27+12+5
Cipher Text	W¦0_2§V_??517û	. . .
??`£YëÃH?¦©_ýT/P|¥EØÃ?j?#_Ó?¸?ú*®ei÷DÞY??ÐgVó,_ã©?_èw)Øê?v_¨??eù¹«ýBI??pùz²?ÌìCÞ_çûÈg_¥åx* Ñ?zb¢jàö®Vj ?CÐ4?Dcq§R?) ýÐ²l __C/ðE9?_Ö/ø7Eg_½?iR?l8Æ?mëç?çaLDbó¢_C_]?n¼¼q«%*U?I 1$_°§?~ÿz?#Ü_²¸7;??PÞxàâ©Bº1tû?_w_E?ì1~M_?aof] ÞAp!bõWóúª2vmççOóy??h_>x+pðáB!þ?`5§w{f®2øè#óíç4?ãó;¥Ðn??î_ë¢ü?¸ì?»~?ç?¡Ðn²âgG?ÿ_ö`_	_T?"9_Ád_%l§P.__	
	?F?)G_Ã]0_c¿2_;?	
	Ê9¤_/_ÎG_7 ·L_SÖ	
	ºD¬_:?é-:_fÑW)_Û	
	Ç&??WÃÏ+?J4×zEtÜ	
	§_·ÙvÞÖ_ 7#ätM?Î	
	??ÙõVõÎ+>V_ÃzkhÄ	
	. . .	

Fig. 10. An example of text encryption.

in addition to the updated key as an input and creates sequences of random numbers for encrypting the blocks B_i.

Updating the key and using plaintext blocks $B_0, B_1, \ldots, B_{i-1}$ to generate random numbers for encrypting the block B_i extremely strengthen the security. They change the state of the random generator and hence greatly correlate the resulting random numbers to the both the plaintext and the current version of the key. The functionality of the random operations that depends on the random numbers changes accordingly. Because the random numbers depend on the key and plaintext, the encryption of each block depends on the previous blocks and the key. The relationship between the plain text and its ciphertext becomes, therefore, complicated since (1) this relationship is random (the key update is random and encryption operations are randomly based) and (2) it depends on both the key and plaintext itself. This greatly immunizes our encryption method against differential analysis techniques for identifying the key.

Fig. 10 shows an example of plaintext encryption. The leftmost column shows the plaintext and its respective ciphertext. The second column shows the original key "p7A12ag798cv6432" and the updated versions. The third column shows a sample of the random numbers sequences that were used during the encryption process. The boldfaced sequences are the ones where the number of pluses equal to number of minuses and therefore a dirty symbol is inserted in the corresponding block. The dirty bytes in the ciphertext are boldfaced and shaded.

4.4. Decryption process

The decryption process decrypts ciphertext using a key. This process performs the same steps of the encryption process, but the random operations are executed in a reverse order and the random subsequences are processed backwards. The random operations are executed in this order: (1) dirty byte extraction (instead of insertion), (2) the random permutation operation, (3) the random shifting operation, and (4) finally the random mutation operation. The random subsequences are processed from the last subsequence Ω_6 to the subsequence Ω_1. Generating the random numbers and updating the keys follow the same steps during the encryption.

The decryption process reads the first block (16 symbols) of the ciphertext. For this block, the decryption process obtains a sequence of random numbers using the key and k random blocks as done during the encryption. It searches for the presence of the dirty symbol in the current block by counting pluses and minuses in the subsequence Ω_6. If the dirty symbol is present, the process (1) removes it at the index $r\%m$, where r is the first non-zero random number in Ω_6 and (2) reads the next symbol, if any, from the remaining ciphertext to complete the current block.

After removing the dirty symbol, the block is ready for decryption. The permutation operation uses the subsequences $\Omega_4\Omega_5$ to reorder the block as described in Section 3.3. It first uses random sequence Ω_5 to reorder the block symbols to match their order before the encryption. Once correctly reordered, the crossover operation uses the random sequence Ω_4 to reverse the effect of this operation on the block during the encryption.

The shift operation obtains the 8-bit representation of the block symbols and organizes them in two dimensional array as described in the encryption section. The operation shifts each row using the first 8 signed random numbers in the subsequence Ω_3. The row shift follows the same steps during the encryption, but the direction of the shift is reversed. The operation left shifts—instead of right shifts—the row i if the sign of the random number r_i is positive and right shifts this row otherwise. After shifting all the rows, the block is reconstructed from the array by concatenating the symbols represented by the columns staring from the leftmost column.

Finally, the mutation operation recovers the plain symbols from their corresponding ciphered symbols in the block. For each ciphered symbol m_i, the mutation operation uses the random number r_i^1, r_i^2 from the subsequences Ω_1 and Ω_2 to index both $SBOX_1$ and $SBOX_2$ and retrieve two symbols ψ and μ. The symbol m_i is XORed with these two symbols yielding the plain text symbol b_i.

After the decryption of the first block, the decryption of the rest, if any, follows roughly the same steps. The only difference is the input to the random generator. As discussed in the encryption pro-

cess, the random number generator uses the latest decrypted blocks and random blocks if needed in addition to the key to produce sequences of signed random numbers.

5. Performance analysis

We test our approach in this section. Two major components on which the effectiveness of the encryption depends and must therefore be tested. The components are the random number generator and the encryption process.

5.1. Encryption/decryption examples

Before testing the proposed encryption method, we show some examples of our method's output. These examples are simple yet give insights about the method's capabilities. Fig. 11 shows the keys, plaintexts, and the ciphertexts. The reader can see the impact of modifying a single bit in the key or the plaintext on the output. This impact is remarkably large. Consider the plaintexts in the first and the second rows in Fig. 11, which differ in only the **boldfaced** and underlined bit. Careful examination of their respective ciphertexts shows that these ciphertexts are largely different. Consider also the second and third rows, which contain the same plaintext but two keys that differ in only one bit. As it could be seen, the encryption method responded to this tiny difference in the key by producing largely different encryptions for the plaintext. The other rows in Fig. 11 lead to similar conclusions.

As the examples may indicate, the output of the method appears as random sequencing of symbols. Additionally and generally speaking, the proposed encryption method seems to respond properly to changes of plaintext or the key. These are very important features of any secure cipher and will be thoroughly tested in the following subsections.

5.2. Test terminologies

We want to test the following two hypotheses: H_0: the output sequence is random and H_1: the output sequence is not random. The statistical test results in accepting either H_0 or H_1 based on some calculated value called **p-value**. The decision of accepting or rejecting H_0 depends on comparing p-value to another specified value called the significance level (or α). Although α can assume any value in [0, 1], the famous choices are 0.05, 0.1, or 0.001.

Given a significance level α, we accept or reject H_0 based on the computed p-value. If p-value is greater than α, accept H_0; reject it otherwise (accept H_1).

5.3. The random generator

We conducted many experiments to analyze the randomness properties of the signed number sequences produced by the random generator. Our experiments include files of 400–2000 symbols collected from the web (Wikipedia). These files are grouped into 5 groups according to their sizes. Group 1 consists of 60 files of size 400 symbols, group 2 consists of 60 files of size 450 symbols, group 3 consists of 30 files of size 500 symbols, group 4 consists of 20 files of size 1000 symbols, and group 5 consists of 20 files of size 2000 symbols. Each file is assigned a different key. We then used the random generator to create three signed number sequences for each file regardless of its size using the assigned key. The first sequence is generated using the original key. To test the impact of updating the key on randomness of the number sequences, we generated two more sequences after updating the key. We also confined the range of the generated signed numbers to be from 0 to 255 (8 bits for each number).

We extracted two additional sequences from each created signed number sequence. The first sequence consists of only the numbers without the sign. The second sequence consists of only the signs after encoding "+" by 1 and "−" by 0, but kept their order as in the original sequence. For instance, if we have the signed number sequence "−12 + 254 + 97-130-9,...", we obtained the two sequences "12, 254, 97, 130, 9, ..." and "01100..." for the numbers and signs respectively.

The original signed number sequence and the two extracted sequences are tested for randomness using three tests chosen from the batteries of randomness tests recommended by the National Institute for Standards and Technology (NIST) (The NIST test suite, 2016; Nechvatal et al., 2010; Vladescu et al., 2014Sys and

Key (16)	Plaintext	Ciphertext
0000000000000000	0000000000000000 0000000000000000	è¶Ú▪ p ÷î£Ã ¹▪ g½µ Ú®0Ò õ±hB ß ▪ C
0000000000000000	0000000**1**00000000 0000000000000000	Fì# Ó ækqpwc7µ-² M)+ $¼iK `¤, È
00**1**0000000000000	0000000**1**00000000 0000000000000000	ü? Í xc ©9þ9À ·1 :°ä ¤CtÓa F Ä
1111111111111111	0000000**1**00000000 0000000000000000	xç¾XãÖhï¾ÆGeR¨▪ `{U N³▪ #] Xh¹Ã ▪
1111111111111111	aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa	Äm¶¿I²Ã9º¯ÓRp ½B-Q)©FAýÔ▪ ▪ 'ú)Ás
0111111111111111	aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa	ï³ÕS ¼4á.w á▪ ½#B7▪ ñôió< Fá !1
77A12BC798EF643D	aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa	5z <½,ù » CÆ&?öÉÍ8Ö▪ nHyîÿ>ûu^ V
77A12BC798EF643D	Bird is flying above house.	¶k ½\x]º \Î▪ ößL¿Þùj0¬û©®
77A12BC798EF643**E**	Bird is flying above house.	l© ´Øòh¥R?çÔDé=-TF Øû ▪ [·

Fig. 11. Examples of the output of our cipher.

Table 5
The results of randomness test for number sequences (without signs).

Sequence (bits)	Runs Test			Monobits			Spectral		
	P-value	Min	Max	P-value	Min	Max	P-value	Min	Max
3200	0.37	0.094	0.87	0.45	0.14	0.45	0.02	0.013	0.056
3600	0.15	0.10	0.18	0.34	0.27	0.43	0.38	0.23	0.51
4000	0.38	0.25	0.56	0.18	0.14	0.29	0.19	**0.0006**	0.23
8000	0.17	0.09	0.31	0.37	0.34	0.40	0.09	0.05	0.16
16,000	0.21	0.1	0.46	0.40	0.24	0.55	0.18	0.12	0.23

Table 6
The results of randomness test for sign sequences.

Sequence (bits)	Runs Test			Monobits			Spectral		
	P-value	Min	Max	P-value	Min	Max	P-value	Min	Max
400	0.87	0.56	0.98	0.74	0.61	0.76	0.35	0.34	0.41
450	0.86	0.56	0.96	0.91	0.78	0.98	0.19	0.09	0.51
500	0.13	0.09	0.28	0.31	0.24	0.37	0.31	0.17	0.50
1000	0.94	0.97	0.92	0.14	0.11	0.22	0.36	0.014	0.62
2000	0.21	0.11	0.45	0.39	0.22	0.48	0.12	0.06	0.20

Riha, 2014). The tests are **Frequency Test** (or **Monobit**), **Runs Test**, and **Discrete Fourier Test** (**Spectral**). We additionally applied the **Runs test** to the signed number sequences using MINITAB software package (MiniTable 17 Statistical Software, 2016).

To apply the NIST's randomness tests, we obtained the 8-bit binary representation for each number. Because every number created by the random generator is an integer between 0 and 255, the binary representation of each number is its binary equivalence. Additional zeros are appended to the most significant positions if the binary representation is fewer than 8 bits. For instance, the binary representations for 192 and 19 are "11000110" and "**000**10011" respectively. (The three zeros appended to the binary representation of 19 appear in bold.) The sign sequences need no further manipulations because they are already in binary representation.

We fed the binary sequences (for the numbers and signs) to each of the three NIST's tests and recorded the significance level of the test in terms of **p-values**. We computed the average of the p-values for all the number sequences and the sign sequences. We also recorded the minimum and maximum p-values.

Table 5 shows the results of randomness test for number sequences. The results are shown in terms of average p-value, minimum, and maximum p-value. Assuming the level of significance is 0.01, the average p-values show no deviation from randomness because all of them are greater than 0.01. The minimum p-values show that one sequence of size 4000 bits failed **Spectral** test because the corresponding minimum p-value is $0.0006<0.01$. Although this sequence failed Spectral test, all of the sequences of this size passed the other two randomness tests. As a result, according to average p-values there is no sufficient evidence that produced number sequences deviate from random.

Table 6 shows the results of randomness test for the signs' sequences. The signs' distribution in signed number sequences does not deviate from randomness because all the average p-values are >0.01. Although some of the minimum p-values (especially in Spectral test) are slightly greater than the significance level, the sign sequences are still statistically random.

We further tested the randomness of the signed number sequences. The number of sequences are 60, 60, 30, 20, 20 for respectively the files of size 400, 450, 500, 1000, 2000. Rather than transforming the numbers in a sequence into binary, we used the signed numbers themselves. The objective is to check if the appearance of the signed numbers in a sequence is random. Table 7 shows the results of running the Runs test using MINITAB software package (MiniTable 17 Statistical Software, 2016). According to the average of p-values and the significance level (0.01), all the sequences passed the Runs test.

Table 7
The results of Runs test for signed number sequences.

Sequence (number of signed numbers)	P-Value		
	Average	Min	Max
400	0.2	0.11	0.68
450	0.49	0.23	0.76
500	0.38	0.015	0.81
1000	0.39	0.30	0.50
2000	0.26	0.15	0.61

Although the test cases may not be sufficiently large, the test results are indicative. They show that generator's outputs do not deviate from randomness for these test cases. Since the test cases are randomly selected, we believe that our conclusions are valid.

5.4. The encryption security

Our encryption method does not directly use keys in the encryption. It encrypts plaintexts using the random numbers generated using the keys. The relation between keys and ciphertexts is hence weak and intractable. This "tiny trace" of keys in ciphertexts makes it infeasible for adversaries to predict keys. Thus techniques that involve ciphertexts analysis are not effective since these ciphertexts involve very limited knowledge about keys. In cryptographical terminologies, our encryption method guarantees a high confusion.

We attribute the high diffusion of our method to the random number generator. The random generator highly depends on keys and plain texts and is thus greatly sensitive to their variations. As discussed in Section 2, variations of keys or plaintexts largely modify the random number sequences and consequently largely influence the functionality of the random operations. The random mutation operation, for instance, is greatly influenced by random numbers; different random numbers index different unicode symbols and hence this operation produces different mutated symbols. The rest of the operations follow suit. The shift and permutation operations use the random numbers to re-position the block symbols. They reposition the symbols differently based on the changes

to the random numbers. Likewise, the functionality of the dirty symbol insertion relies on the random numbers.

Known effective attack methods such as Differential Cryptanalysis (Biham and Shamir, 1991; Biham and Shamir, 1993), Linear Cryptanalysis (Matsui, 1994), Truncated Differentials (Knudsen, 1994), Boomerang Attacks (Wagner, 1999), Impossible Differentials (Sung et al., 2904) and others (Dewu and Wei, 2010; Biham and Dunkelman, 2017) use analytical techniques that involve ciphertexts to predict keys. However, our ciphertexts have too limited trace of keys to help predict these key. Furthermore, identifying random numbers provides no help to predict the key because the random number generation process is irreversible.

5.4.1. Bit-level randomness test

We conducted many experiments to test the randomness properties of the output of our encryption method. Without losing the generality of the test, we confined the SBOX's contents to only 256 unicode symbols. This allows for representing the output symbols in 8 bits. Given this confine, we tested the output at the bit level.

We used the following three sets of data to test the randomness at the bit level (Vladescu et al., 2014; Soto, 2017).

- *Key Avalanche Test*. The objective of this data set is to examine the sensitivity of our algorithm to changes in the key.
- *Plaintext Avalanche Test*. The objective of this data set is to examine the sensitivity of our algorithm to changes in the plaintext.
- *Plaintext/Ciphertext Correlation*. The objective is to study the correlation between plaintext-ciphertext pairs.

We prepared these sets of data as described in Soto (2017). Firstly, to study the sensitively of our algorithm to the key change, we created and analyzed 680 sequences of size 16,000 bits each. We used a 128-bit plaintext of all zeros and 20 random keys each of size 128 bits (16 bytes). Each sequence was created by concatenating 125 derived blocks constructed as follows. Each derived block is created by XORing the ciphertext created using the fixed plaintext and the 128-bit key with the ciphertext created using the fixed plaintext and the perturbed random 128-bit key with the ith bit changed, for $1 \leqslant i \leqslant 128$.

Secondly, to analyze the sensitivity to the plaintext changes, we created and analyzed 680 sequences of size 16,000 bits each. Each sequence was created by concatenating 125 derived blocks constructed as before. Each derived block is created by XORing the ciphertext created using the 128-bit key and the plaintext with the ciphertext created using the 128-bit key and the perturbed random 128-bit plaintext with the ith bit changed, for $1 \leqslant i \leqslant 128$.

Thirdly, to study the correlation of plaintext-ciphertext pairs, we constructed 680 sequences of size 64,000 bits per a sequence. Each sequence is created as follows. Given a random 128-bit key and 500 random plaintext blocks, a binary sequence was constructed by concatenating 500 derived blocks. A derived block is created by XORing the plaintext block and its corresponding ciphertext block.

Tables 8 and 9 show the randomness test results for key avalanche, plaintext avalanche, and plaintext-ciphertext correlation. The results are presented in terms of number of successes, failures, and the rate of success. A sequence passes the randomness test if its corresponding p-value is greater than 0.01. The success rate is the percentage of the passed sequences to the total number of sequences (680). Table 8 shows that the sequences have passed the Runs Test, Monobits, and Spectral with success rate of 90%, 89.3%, and 85.8% respectively. Table 9 shows that the sequences have passed the Runs Test, Monobits, and Spectral with success rate of 82.4%, 85.4%, and 79.6% respectively. Table 10 shows that the sequences have passed the Runs Test, Monobits, and Spectra with success rate of 93.09%, 95.1%, and 88.8% respectively.

Table 8
Key avalanche test results.

Test	Successes	Failures	Rate of success
Runs Test	612	68	90%
Monobits	607	73	89.3%
Spectral	584	96	85.8%

Table 9
Plaintext avalanche test results.

Test	Successes	Failures	Rate of Success
Runs Test	560	120	82.4%
Monobits	581	99	85.4%
Spectral	541	139	79.6%

Table 10
Plaintex-ciphertext correlation test results.

Test	Successes	Failures	Rate of Success
Runs Test	633	47	93.09%
Monobits	647	33	95.1%
Spectral	604	76	88.8%

Table 11
The results of Runs test for ciphertext symbols.

Files		P-Value		
Size	number	Average	Min	Max
48	30	0.88	0.80	0.97
437	60	0.61	0.52	0.69
524	60	0.72	0.64	0.78
1000	50	0.54	0.49	0.62
2000	30	0.77	0.72	0.81
2859	10	0.66	0.62	0.72
3000	30	0.71	0.70	0.76
10000	30	0.81	0.74	0.87
20000	12	0.89	0.79	0.95

Although the input sequences are not sufficiently large, the test results are promising. More than 80% of the sequences passed the randomness test for all of the three sets of data except Spectral test fell under 80% for plaintext avalanche. For some tests, even more than 90% of the sequences passed the randomness test. Most of the failures happened when the input blocks contain large subsequences of identical symbols. As a future work, we will consider different ways to enhance the randomness properties of the method (e.g. making the encryption process iterative).

5.4.2. Symbol-level randomness test

We tested the output of our method at the symbol level. We selected 112 files of different sizes from the Wikipedia. We in addition created 200 random files of different sizes. The random files are created by using the computer random generator to generate random integers in the range [0, 255] and concatenate their respective unicode symbols. All the random files of sizes $\geqslant$1000. We encrypted the 312 files using our method and tested the randomness of the resulting ciphertexts. For each ciphertext, we created a sequence of integers by finding the unicode index of each symbol. For instance, the unicode index for the symbol "A" is 65. We then applied Runs test to each sequence using MINITAB statistical package. Table 11 shows the results of Runs test.

As Table 11 shows, all the p-values are significant (> 0.01). This outcome indicates that all of the ciphertexts do not deviate from

the randomness regardless of whether the plaintext file is random or not.

6. Conclusions and future work

We are presenting a very illusive encryption technique. The key is used in random number generation greatly isolating the key from the encryption process. The key is also continuously updated. Random mutation provides radical departure from the current state of the code. Shifting, permutations and crossover provide random but limited alterations on the coded pattern. Their alterations are more local and they still keep the code within the vicinity of its previous state. Mutations make big and radical changes on the structure of the encrypted text. The continuous updating of the key makes it very difficult for analytical methods to discover a hidden pattern and connect it with some key.

The randomness tests conducted on the ciphertext are of several different natures; with or without sign; bit level or symbol level. The samples of text are diverse and come from different resources. Some of these test are in time domain, others are in frequency domain. The time domain behaviors along with transformations for the frequency domain behaviors are contributing to a comprehensive test. The randomness and the non-cyclic behavior are mostly desired behaviors. The zero crossing and the uniformity of the random generated numbers are tested.

On the other hand, the encryption process itself was tested in terms of sensitivity to changes in key, plaintext, and plaintext/ciphertext correlation. From the results it is clear that the results of Spectral test are the weakest. This is a general problem since our random number generators depend a lot on the LCG generator that has the problem of generating numbers that are organized in clear successive two dimensional planes for $x(t)$ and $x(t-1)$. This problem can be overcome if our original random numbers generators RNG change its dependence on the mod() function and start introducing more complex function. Of course, that would introduce more complexity and cost.

Furthermore, the testing for sensitivity to keys' change and plaintext's change preserved the needed randomness in almost all the three tests we applied. The tests passed the preset p-values of 0.01 in about 90% of the cases except in the spectral test. Despite the minor alterations in the key and in the plaintext, the ciphertexts were still maintaining its randomness characteristics. The tests on the bit level and on the symbol level showed high success rates. However, as mentioned earlier, longer sequences of bits/symbols would reflect more reliable evidences about the performances of the random number generator we have.

Using other sources of random number generators such as using the natural noise or input from physical phenomena could be a more realistic source of random numbers. Random.org uses radio receivers to collect random radio signals from the space and use them as source of randomness. It would be very interesting, as future work, to use these sources and repeat the simulation we had in this work.

In general, a new elusive random number generation technique has been used. The direct influence of the encryption key on the ciphertext was minimized. It would be very hard to analyze the encryption patterns and figure out the key pattern. We still hope for further improvement on the performance by trying alternative resources for the RNG other than the original PRNG we are depending on. This will be the topic of future work and hopefully much better results will be achieved.

References

Biham, E., Dunkelman, O., 2017. Techniques for Cryptanalysis of Block Ciphers. Information Security and Cryptography. Springer-Verlag, Berlin Heidelberg.

Biham, E., Shamir, A., 1991. Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. 4, 3–72.

Biham, E., Shamir, A., 1993. Differential Cryptanalysis of the Data Encryption Standard. Springer-Verlag.

Biham, E., Anderson, R., Knudsen, L., 2000. Serpent: A Proposal for the Advanced Encryption Standard.http://cryptosoft.net/docs/Serpent.pdf. Porposal for the Advanced Encryption Standard

Burnwick, C., et al. 1999. The Mars Encryption Algorithm. IBM.

Daemen, J., Rijmen, V., 2001. Advanced Encryption Standard(AES).http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf/, November 2001.

Dewu, X., Wei, C., 2010. A Survey on Cryptanalysis of Block Ciphers. In 2010 International Conference on Computer Application and System Modeling (ICCASM 2010), volume 8, pages 218–220, Oct 2010.

Kamalakannan, V., Tamilselvan, S., 2015. Security enhancement of text message based on matrix approach using elliptical curve cryptosystem. Procedia Mater. Sci. 10, 489–496.

Knudsen, L., 1994. Truncated and Higher Order Differentials. Fast Software Encryption, Springer LNCS 1008, 196–211.

L'Ecuyer, Pierre, 2005. On the Xorshift random number generators. ACM Trans. Model. Comput. Simul. (TOMACS) 15 (4), 346–361.

L'Ecuyer, Pierre, 2012. Random Number Generation. In: Gentle, James E., Karl Härdle, Wolfgang, Mori, Yuichi (Eds.), Handbook of Computational Statistics, Springer Handbooks, chapter 3. Springer, Berlin Heidelberg, pp. 35–71.

Marsaglia, G., 2003. Xorshift random number generators. J. Stat. Softw. 14 (8), 1–6.

Matsui, M., 1994. Linear Cryptanalysis Method for DES Cipher. In: Advances in Cryptology-EuroCrypt'93. In: Helleseth, Tor (Ed.), . Lecture Notesin Computer Science, vol. 765. Springer-Verlag, Berlin, pp. 386–397.

MiniTable 17 Statistical Software. Website, 2016.www.minitab.com.

Nagaraj, S., Raju, D.S.V.P., Bhamidipati, K., 2013. Randomized approach for block cipher encryption. In: Satapathy, S.C. et al. (Eds.), Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA). Springer-Verlag, Berlin Heidelberg, pp. 551–558.

Nechvatal, J., Rukhin, A., Soto, J., et al. 2010. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Special publication 800-22, National Institute of Standards and Technology (NIST).

Preneel, B. De Canniére, C., 2006. Trivium – A Stream Cipher Construction Inspired by Block Cipher Design Principles. In: Proceedings of the 9th International Conference on Information Security – ISC, pages 171–186, Samos Island, Greece, August 2006.

Rose, Greg, 1998. A stream cipher based on linear feedback over GF(2^8). In: Boyd, Colin, Dawson, Ed. (Eds.), Information Security and Privacy, vol. 1438. Springer, Berlin Heidelberg, pp. 146–155.

Shub, M., Blum, L., Blum, M., 1986. A simple unpredictable pseudo-random number generator. SIAM J. Comput. 15 (2), 364–383.

Sidney, R. Rivest, R., Robshaw, M. Yin, Y. 1998. The RC6 Block Cipher.http://people.csail.mit.edu/rivest/pubs/RRSY98.pdf.

Singh, L.D., Singh, K.M., 2015. Implementation of text encryption using elliptic curve cryptography. Procedia Comput. Sci. 54, 73–82.

Soto, Juan Jr. 2017. Randomness Testing of the AES Candidate Algorithms.http://csrc.nist.gov/archive/aes/round1/r1-rand.pdf, Accessed 2017.

Stalling, W., 2016. Cryptography and Network Security: Principles and Practices. Pearson.

Steef, A., Shamma, M.N., Alkhatib, A., 2015. RSA algorithm with a new approach encryption and decryption message text by ASCII. Int. J. Cryptography Inf. Security (IJCIS) 5 (3), 23–32.

Sung, J., Kim, J., Hong, S., et al., 2904. Impossible Differential Cryptanalysis for Block Cipher Structures, volume 2904 of Lecture Notes in Computer Science. Springer, Berlin Heidelberg, pp. 82–96.

Sỳs, M., Rĭha, Z., 2014. Faster Randomness Testing with the NIST Statistical Test Suite. In: Schaumont, P., Chakraborty, R.S., Matyas, V. (Eds.), Security, Privacy, and Applied Cryptography Engineering, volume 8804 of Lecture Notes in Computer Science. Springer, Cham, pp. 272–284.

The NIST test suite. Website, 2016.http://csrc.nist.gov/groups/st/toolkit/rng.

Vladescu, F., Gheorghe, L., Duta, C., Mocanu, B., 2014. Randomness evaluation framework of crypotographic algorithms. Int. J. Cryptography Inf. Security (IJCIS) 4 (1), 31–49.

Wagner, D., 1999. The boomerang attack. Fast Software Encryption, Springer LNCS 1636, 156–170.

Whiting, D., Schneier, B., Kelsey, J., et al. 1998. Twofish: A 128-Bit Block Cipher. Technical report,https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf.