International Journal of Innovative Computing, Information and Control
Volume **16**, Number **1**, February **2020**
ICIC International ©2020 ISSN 1349-4198
pp. 257–282

INTELLIGENT CONVOLUTIONAL MESH-BASED ENCRYPTION TECHNIQUE AUGMENTED WITH FUZZY MASKING OPERATIONS

Muhammed Jassem Al-Muhammed[1] and Raed Abu Zitar[2]

[1]Faculty of Information Technology
American University of Madaba
Madaba 11821, Jordan
m.almuhammed@aum.edu.jo

[2]College of Engineering and Information Technology
Ajman University
Ajman 0346, U.A.E
r.abuzitar@ajman.ac.ae

Received April 2019; revised August 2019

Abstract. *In this digitally-led world, users either digitally store or transmit their sensitive information. This information is vulnerable to all types of threats unless properly secured. In this paper, we address the problem by presenting a new encryption technique. The proposed technique introduces several processes and operations that make it extremely effective. First, the substitution process makes deep changes to its input, thereby greatly weakening the relationship between the input and the output. Second, the diffusion processes highly increase the avalanche effect. Third, the technique intelligently inserts random noises in the encryption to further increase the ciphertext confusion. Fourth, the technique presents highly effective encoding operation that adds further diffusion to the ciphertext. The conducted experiments on our prototype implementation showed that our technique is effective and passed important security tests.*
Keywords: Encryption technique, Convolutional mesh-based mapping, Substitution/ Inverse substitution methods, Block encoding/decoding, Block diffusion, Key expansion

1. **Introduction.** In the information revolution era, we are overwhelmed with tremendous amount of information. Individuals digitally keep or transmit information about almost every aspect of their lives. This information is likely to be very sensitive and probably is life-threatening if learned by unauthorized people. As such, ensuring the security of this information is the highest priority for every information owner whether individual or organization.

Cryptography presents itself as the de-facto mechanism for ensuring the highest degrees of information security. Searching for reliable techniques, many encryption methods have been proposed (e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 18, 19, 21]). All these techniques, but Rivest-Shamir-Adleman (RSA) [9], are symmetric in a sense they use one secret key for both encryption and decryption. RSA is, however, an asymmetric technique because it uses two different keys: a public key for encryption and a private key for decryption. Regardless of using a single key or two keys, all these techniques encrypt plaintext blocks using deterministic methods that involve operations such as substitution, shifting, permutations, and mathematical manipulations. For instance, the Advanced Encryption Standard, AES [6], encrypts blocks of plaintext using a fixed number of rounds, each round handles the input by applying four operations: substitution, row

DOI: 10.24507/ijicic.16.01.257

shifting, column mixing, and add round key. The approach proposed in [21] has a bit different processing model. It encrypts messages by means of two keys each of which is a number from 0 to 7. The first key works at the bit level, which is used to split the byte (8 bits) into two parts and flips only the bits of the second part. The second key is used to split the whole message into intervals whose length equals the second key. Each interval will be handled using the first key.

We described in a previous paper [1] a technique for information encryption. This technique strongly relies on a mesh-based mapping operation, which maps plaintext symbols to signed integers called directives (directives are discussed in later sections). It also offers additional effective operations for key expansion, directive masking, and final directive substitution. Security testing has shown that this technique is effective as it passed important security test cases.

This paper extends the technique [1] further by suggesting fundamental extensions and performing further testing and simulations. In particular, it significantly improves the final-directive substitution by basing this substitution on much "trickier" techniques that involve both randomness and masking operations. Our proposed directive substitution operation enables the encryption technique to gain further confusion and diffusion. As such, this paper makes the following contributions.

1) We define an effective directive substitution operation that greatly improves the provided security of the algorithm (Section 8).
2) We perform extensive simulations to empirically tune important parameters of the encryption technique (Appendix A – provided as an external file due to space restriction).
3) We further tested the randomness of the encryption technique and random generator by performing additional randomness tests (Section 10).
4) We conduct further analysis of the random number generator (Appendix B – provided as an external file).

2. Substitution and Inverse-Substitution Methods.

Substituting a block is the process of replacing its symbols with new ones using a substitution table called S-Box [6]. The S-Box as described in [6] (see Figure 1(a)) is a 16×16 lookup table with 256 entries that represent all the possible combinations of the byte. As described in [6, 11], to substitute a symbol, we divide the 8 bits that represent the symbol into two halves (4 bits each). The left 4 bits index the row of the table and the right 4 bits index the column. The indexed value is the substitute for the symbol. For instance, to substitute the symbol "M" ("4D" in Hex), the left 4 bits (i.e., "4") index the row and the right 4 bits (i.e., "D") index the column, yielding "E3" (or "ã").

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(a) The S-Box

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

(b) The Inverse S-Box

FIGURE 1. The data used in the substitution process

To obtain the original symbols, S-Box has an Inverse S-Box. Figure 1(b) shows the Inverse S-Box for the S-Box in Figure 1(a). Substituting a symbol using Inverse S-Box is identical to substituting a symbol using S-Box.

2.1. **The Substitute method.** The proposed **Substitute** method uses the same S-Box [6], but offers a fundamentally different substitution mechanism. First, our substitution method uses the outcome of the most recently performed substitution to substitute the current symbol. Second, our substitution operation consists of two substitution actions: **Forward** and **Backward**. Figure 2 shows the steps of the substitution process. The Forward Substitute takes the input block $k_1k_2\ldots k_n$ and returns the substituted block T ($\mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$). Referring to Figure 2, the Forward Substitute substitutes the first symbol k_1 with a new symbol ($\mathcal{T}_1$) using the S-Box. For the symbols k_i ($i = 2, 3, \ldots, n$), the Forward Substitute XORes the most recent substitution outcome $\mathcal{T}_{i-1}$ with the to-be-substituted symbol k_i. The outcome of the XOR operation $K_i^\star$ is substituted using the S-Box to produce the new symbol $\mathcal{T}_i$ in the output block T.

Forward Substitute
Input: $k_1k_2\ldots k_n$
Output: substituted block T
$\mathcal{T}_1$= Substitute (k_1)
T = T + $\mathcal{T}_1$
FOR i = 2 to n **DO**
$K_i^\star = \mathcal{T}_{i-1} \oplus K_i$ $\mathcal{T}_i$ = Substitute($K_i^\star$)
T = T + $\mathcal{T}_i$ **RETURN** T

Backward Substitute
Input: $\mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$
Output: substituted block $\mathcal{W}$
w_n= Substitute ($\mathcal{T}_n$)
$\mathcal{W}$ = $\mathcal{W}$ + w_n
FOR i = n-1 to 1 **DO**
$K_i^\star = w_{i+1} \oplus \mathcal{T}_i$ w_i = Substitute($K_i^\star$)
$\mathcal{W}$ = $w_i + \mathcal{W}$ **RETURN** $\mathcal{W}$

FIGURE 2. The Substitute operation

The output of the Forward Substitute T ($\mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$) is passed to the Backward Substitute. The Backward Substitute has the same functionality as the Forward except that it starts from the end of the input. Therefore, as Figure 2 shows, the Backward Substitute substitutes the last symbol $\mathcal{T}_n$ in the input first with new symbol (w_n) using the S-Box. For the symbols $\mathcal{T}_i$ ($i = n-1, n-2, \ldots, 1$), the Backward Substitute first XORes this symbol ($\mathcal{T}_i$) with the most recent substitution outcome (w_{i+1}). The outcome of the XOR operation $K_i^\star$ is substituted to produce the new symbol w_i in the output block $\mathcal{W}$.

2.2. **The Inverse-Substitute method.** The **Inverse-Substitute** method reverses the impact of the **Substitute** method. The functionality of the Forward Inverse-Substitute and the Backward Inverse-Substitute are defined in Figure 3. The Backward Inv-Substitute is executed first then the Forward operation. Furthermore, the Inverse-Substitute uses the Inverse S-Box (rather S-Box) when substituting a symbol.

Backward Inv-Substitute
Input: $w_1\, w_2\, \ldots\, w_n$
Output: substituted block T
$\mathcal{T}_n$= Substitute (w_n)
$T = \mathcal{T}_n + T$
FOR i = n-1 to 1 **DO**
t_i = Substitute(w_i) $\mathcal{T}_i = t_i \oplus w_{i+1}$
$T = \mathcal{T}_i + T$ **RETURN** T

Forward Inv-Substitute
Input: $\mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$
Output: substituted block K
k_1= Substitute ($\mathcal{T}_1$)
$K = K + k_1$
FOR i = 2 to n **DO**
S_i = Substitute($\mathcal{T}_i$) $k_i = S_i \oplus \mathcal{T}_{i-1}$
$K = K + k_i$ **RETURN** K

FIGURE 3. The Inverse-Substitute operation

The Backward Inv-Substitute starts from the end of the input. Therefore, it substitutes the last input symbol w_n using the inverse S-Box to yield the output symbol $\mathcal{T}_n$. For the symbols w_i $(i = n-1, n-2, \ldots, 1)$, the Backward Inv-Substitute first substitutes the input symbol w_i and then XORes the outcome t_i with the most recently processed input symbol w_{i+1}. The output of the Backward Inv-Substitute $\mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$ is then passed to the Forward Inv-Substitute. The Forward Inv-Substitute has the same functionality as the Backward except that it starts from the first input symbol rather than from the last one. Therefore, it first substitutes the input symbol $\mathcal{T}_1$ using the inverse S-Box to yield the output symbol k_1. For the remaining input symbols $\mathcal{T}_i$ $(i = 2, 3, \ldots, n)$, the Forward Inv-Substitute first substitutes the input symbol $\mathcal{T}_i$ and then XORes the outcome S_i with the most recently processed input symbol $\mathcal{T}_{i-1}$.

3. **Key Expansion Method.** The key expansion method extends encryption keys to any arbitrary length. The method uses two operations: Substitute operation and Manipulate operation. The Substitute operation substitutes the symbols of the input key as described in Section 2. The output of the Substitute operation, say $t_1t_2\ldots t_n$ is then passed to the Manipulate operation for further processing. The Manipulate operation largely distorts its input $(t_1t_2\ldots t_n\ )$ by modifying both individual symbols and the structure of the input block. To process a symbol t_i, the Manipulate operation applies one of the two actions $Flip^{LH}(t_i)$ and $FlipSwap^{RH}(t_i)$ to this symbol. The action $Flip^{LH}(t_i)$ makes changes to the symbol t_i by flipping its left half bits. The action $FlipSwap^{RH}(t_i)$ makes dual changes to the input: it makes changes to the symbol t_i by flipping the right half bits of t_i and it moves the resulting symbol to a position determined by the lookahead symbol t_{i+1} (The symbol t_{i+1} is the successor of the symbol t_i in the input).

To select which of the two actions is applied to an input t_i, the Manipulate operation uses a fuzzy selection mechanism that relies on the lexical order of its input symbols. Suppose, we want to manipulate the input symbol t_i. Let T be the number of the so-far processed symbols and P be the number of times in which $t_i \le t_{i+1}$ holds. Based on this, we define the likelihood variable $LTE = P/T$, where LTE ("Less Than or Equal") is the ratio in which the condition $t_i \le t_{i+1}$ holds. Using this likelihood variable, we define our fuzzy *Manipulate* operation in Figure 4. As Figure 4 shows, the amount of the bias toward any of the two actions changes as the values of LTE change. When LTE increases (decreases), the likelihood of executing the *Flip* action increases (decreases) over the likelihood of executing *FlipSwap* action. Hence, any increase (or decrease) in the value of LTE does change the likelihood of executing an action over the other, but the choice of which action to execute is random since it depends on the random value γ_i.

Manipulate (t_i)
  Generate a random number $\gamma_i \in (0, 1)$
  If $(\gamma_i \le LTE)$ THEN execute $Flip^{LH}(t_i)$ action
  Else execute $FlipSwap^{RH}(t_i)$ action

FIGURE 4. The fuzzy process for action selection

After defining the two operations, we delineate the key expansion method in Figure 5. The process takes a key of length n as an input and returns an extended key (of arbitrary length) as an output. As Figure 5 shows, the key expansion process consists of the steps 1 through 10. The symbols of the input key are first substituted with new symbols x_i's using the Substitute operation. Next, the new symbols x_i's are manipulated using the Manipulate operation (Figure 4). To define the amount of the bias LTE (line 6), we

```
Input: Key=k1k2 ...kn
Output: Expanded-Key = Key
1.  Let L = Key, T = 1, P =0
    /*T=total number of processed symbols and P = number of times in which
    the current input symbol lexically comes before the lookahead symbol*/
2.  x1x2 ... xn←Substitute (L)
3.  For i=1 to n Do
4.    If (xi ≤ xi+1) P ++ //increment P
5.    T ++
6.    LTE = P/T
7.    S = S+ Manipulate (xi)
8.  Expanded-Key = Expanded-Key + S
9.  If desired length not reached yet, L = S, GOTO 2
10. Return Expanded-Key
```

FIGURE 5. The algorithmic steps for the key expansion method

always increment the number of processed symbols T while incrementing P only when the currently processed symbols x_i lexically equals or is before the lookahead symbols x_{i+1}. Observe, step 9 ensures that the key expansion method repeats until the key is expanded to the desired length.

4. **Key-Based Random Number Generator.** The random number generator uses the encryption key to generate sequences of random numbers that provide input for the encryption (decryption) process. Ideally, the generator should possess two properties to be effective in the security field: 1) highly responsive to the changes of the key regardless of the magnitude of the change and 2) pass randomness tests. Figure 6 shows the algorithmic steps of our generator that ensure two properties for ideal random generator.

```
Input: Key
Output: sequence of key-based numbers with an arbitrary length.
1. Extend the key to 64 symbols to get the Seed.
2. Seed = Substitute (Seed)
3. Seed = Flip^R (Seed, n, m) /*flip the right n bits in the symbol Seed[m]*/
4. Seed = Shift^L(Seed, k)    /* circular left shift symbols of Seed k positions*/
5. For i=1 to |Seed|          /*|Seed| is the length of Seed*/
6.   SUM += 256^(|Seed|-i) × (INT) Seed[i] /*Seed[i] is the symbol at index i*/
7. RAND = SUM mod P   /*mod is module operation and P is the range of the numbers*/
8. If More numbers needed, GO TO 2
```

FIGURE 6. The algorithmic steps for the key-based number generator

In step (1), the generator extends the key to 64 symbols so that the period of the generator increases. The steps (2)-(7) define the major functionality of the generator. Step (2) uses the Substitute method (Section 2) to propagate the changes in any bit of the key to the other symbols – this ensures that a small change results in highly different seed. Step (3) declares the operation $Flip^R$, which flips the right n bits of the symbol at index m. The values of n and m are respectively the integer values for the seed symbol at index 0 and the seed symbol at index 1 (i.e., $n =$ (INT) Seed [0] and $m =$ (INT) Seed [1]). Step (4) declares the operation $Shift^L$, which circularly left shifts the symbols of the resulting Seed by k positions. The value of k is the integer value for Seed [2]. Clearly the operations (3) and (4) change the Seed, allowing the Substitute operation (2) to more effectively change the Seed. Steps (5) and (6) sum up the symbols of the resulting Seed by multiplying the integer value of the Seed symbol at index i with the power of 256. The reason for taking the power of 256 is that radix for the symbols 0 to 255 is 256. Therefore, this summation never yields the same Sum value for different Seeds.

5. **Block Diffusion Method.** The sensitivity of encryption techniques to the input changes is a very important property. The encryption technique is highly sensitive to an input change if a tiny change to the input causes tremendous changes to the output. This property has a large impact on one of the most important security measures: the avalanche effect.

We propose a diffusion method that provides our encryption technique with an extremely high avalanche effect. The diffusion method defines two methods: 1) the Diffuse (DIF), which amplifies the input's change and makes this change largely affect all the output's symbols and 2) the Inverse-Diffuse (INV-DIF), which reverses the impact of the DIF to obtain the original block. Before we explain the DIF and INV-DIF methods, we introduce two operations that are used in the DIF and INV-DIF. These two new operations are Distort and Inverse-Distort.

5.1. **The Distort operation.** The Distort operation (DIS) makes drastic changes to individual symbols of the input block as well as to the structure of this block (Changing the structure of the block means breaking the original order of its symbols). The Distort operation defines 56 different distort actions. Table 1 shows the actions, their inverse, and a simple description of each action. The actions are of two types: unary and binary. The unary actions take a single input symbol while the binary actions take two input symbols.

Table 1. The distort actions

	Action	Description	Inverse Action	Description
Unary operations	***SL_n***	Circularly shifts the bits of the input symbol *n* positions to the left (*n*=1, 2, …, 8)	***SR_n***	Circularly shifts the bits of the input *n* positions to the right (*n* =1, 2, …,8)
	SR_n	Circularly shifts the bits of the input *n* positions to the right (*n* =1, 2, …,8)	***SL_n***	Circularly shifts the bits of the input symbol *n* positions to the left (*n*=1, 2, …, 8)
	ML_n	Mutates (flips) the left *n* bits of the input symbol (*n* =1, 2, …, 8)	***ML_n***	Mutates (flips) the left *n* bits of the input symbol (*n* =1, 2, …, 8)
Binary operations	***CLL_n***	Crosses over the left *n* bits of the first input symbol with the left *n* bits of the second input symbol (*n*=1, 2, …,8)	***CLL_n***	Crosses over the left *n* bits of the first input symbol with the left *n* bits of the second input symbol (*n*=1, 2, …,8)
	CRR_n	Crosses over the right *n* bits of the first input symbol with the right *n* bits of the second input symbol (*n*=1,2,…,8)	***CRR_n***	Crosses over the right *n* bits of the first input symbol with the right *n* bits of the second input symbol (*n*=1,2,…,8)
	CRL_n	Crosses over the right *n* bits of the first input symbol with the left *n* bits of the second input symbol (*n*=1,2, …, 8)	***CRL_n***	Crosses over the right *n* bits of the first input symbol with the left *n* bits of the second input symbol (*n*=1,2, …, 8)
	CLR_n	Crosses over the left *n* bits of the first input symbol with the right *n* bits of the second input symbol (*n*=1,2,…,8)	***CLR_n***	Crosses over the left *n* bits of the first input symbol with the right *n* bits of the second input symbol (*n*=1,2,…,8)

Each of the unary actions SL_n and SR_n takes a single input symbol and circularly shifts the bits of this symbol n bits to respectively the left or the right. The unary action ML_n mutates (flips) the left n bits of its input symbol.

The binary actions take two input symbols. The CLL_n action crosses over (exchanges) the left n bits of the first argument with the left n bits of the second argument. The CRL_n action exchanges the right n bits of the first argument with the left n bits of the second argument. Note for the binary actions, when n equals the total number of the bits that represent the symbol, the impact of these actions is to permute the positions of two arguments in the block. Therefore, the binary distort actions have dual impact on their inputs: distort the individual symbols or alter the structure of the input block.

These distort actions are placed in a lookup table. Each row of the table consists of two columns, where the action is stored in the first column and its inverse is stored in the second. In order to make the selection of any of the entries random, we randomly reorder the actions in the lookup table using a sequence of random numbers obtained from our random generator. The reordering is performed by swapping the contents of the row i with the contents of the row r_i (r_i is random number).

The Distort operation (DIS) handles its input block in two passes: forward (left to right) and backward (right to left). Let $b_1b_2\dots b_n$ be a plaintext block. The forward pass handles the block as follows. The first symbol b_1 in the input block stays without processing. For all the symbols b_i ($i > 1$), the operation applies one of the 56 distort actions to the input. The selection of which action (be unary or binary) to apply to the symbol b_i is fully determined by the most recently processed input symbol b_{i-1}. More specifically, let $c_1c_2\dots c_{i-1}$ be the so-far output block that results from processing the input symbols $b_1b_2\dots b_{i-1}$. To select an action for processing the next input symbol(s), the forward pass uses the most recently processed input symbol b_{i-1}. We basically compute the index of the distort action by module operation: $j = \text{Val}\ (b_{i-1}) \bmod 56$, where Val (b_{i-1}) is the decimal code of the symbol b_{i-1} and "mod 56" since we have 56 actions. The action at the calculated index j is applied to the symbol b_i or the two symbols b_i and b_{i+1} based on whether the selected action is unary or binary. If the selected action is unary, this action processes the symbol b_i and yields the output symbol c_i. If the selected action is binary, this action processes the two input symbols b_i and b_{i+1} and yields the output symbols c_i and c_{i+1}. In case that the action is binary and the input block has only one unprocessed input symbol, the action is ignored and the input symbol is appended to the output block (without processing).

The backward pass performs identical processing except that it starts from the end of the input $c_1c_2\dots c_n$. Therefore, the first symbol to start with is c_n, which is left unprocessed. For all the symbols c_i ($i < n$), the backward pass uses the most recently processed input symbol c_{i+1} to select the distort action for processing the input symbol c_i or the input symbols c_i and c_{i-1} depending on whether the selected action is unary or binary. Additionally, to highly secure the output of the Distort operation, the backward pass seals the rightmost output's symbol by XORing it with a random number obtained from the random generator.

5.2. **The Inverse-Distort operation.** The Inverse-Distort operation (INV-DIS) reverses the changes made by the Distort operation (DIS). In other words, INV-DIS operation reverses the impact of DIS operation and recovers the original block. Like DIS operation, INV-DIS operation handles its input in two passes: forward and backward. Let $b_1b_2\dots b_n$ be an input block to INV-DIS operation. The INV-DIS starts with the **backward** pass. In this pass, INV-DIS first unseals the rightmost input symbol b_n to obtain the new symbol c_n. The symbol c_n is appended to the output without further processing. Thus, the so-far output list contains only "c_n". For the input symbols b_i ($i < n$), the INV-DIS operation uses the most recently appended symbol to the output c_{i+1} to determine which inverse action (Table 1) to apply to the input block. Determining which inverse action to apply is done exactly as done in the DIS operation.

The output of the backward pass is passed to the **forward** pass. This pass performs identical steps as the backward except it starts from the leftmost symbol of the input block. The leftmost symbol c_1 is appended directly to the output without processing. For all symbols c_i ($i > 1$), the forward pass uses the most recently appended symbol to the output, say d_{i-1}, to determine the inverse action to apply to the input symbol c_i or to the input symbols c_i and c_{i+1} depending on whether the inverse action is unary or binary.

5.3. **The DIF method.** This method consists of two operations: 1) Substitute operation and 2) Distort operation (DIS). The Substitute operation executes first. Based on definition of the Substitute operation (Section 2), when a symbol b_i changes, the forward Substitute ensures that this change affects not only the substitution outcome of b_i, but also the substitution outcome of all the symbols b_j ($j > i$). In addition, the backward Substitute ensures that the change in b_i affects all the symbols b_j ($j < i$). As a result, the change actually impacts every single symbol in the block. The output of the Substitute operation is passed to the Distort operation. This operation changes the individual symbols and the structure of the input block.

5.4. **The INV-DIF method.** The INV-DIF method (Inverse Diffusion) reverses the impact of the Diffuse operation (DIF). This operation uses both the Inverse-Substitute operation and Inverse-Distort (INV-DIS). These two operations are executed in the following sequence: **INV-DIS** then **Inverse-Substitute**.

6. **Block Encoding.** The block encoding process transforms plaintext blocks into a new representation in which the relationship to the original plaintext block is melted. Figure 7 shows the logic of this process. The process consists of two operations: 1) *Block Mapping*, which consists of two actions *Initial Move* and *Mesh-Based Mapping* and 2) *Block Manipulation*, which consists of two actions *Masking* and *Permutation*.

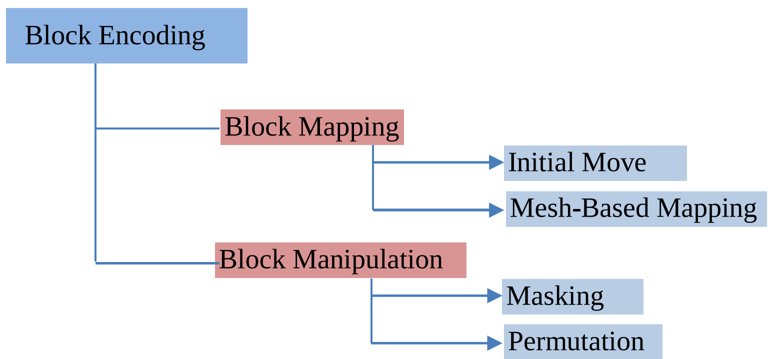


FIGURE 7. The block encoding process

The four actions in Figure 7 require appropriate input to function. We, therefore, define the operational knowledge $\mathcal{P}$ in Figure 8 to provide such an input. As the figure shows, the operational knowledge is a binary tree whose edges are labeled with 0's or 1's. The tree is divided into three levels, each of which provides the necessary input for one or more of the four actions. Each path in the tree leads to some input value for an action. For instance, the path labeled with "000" leads the input value L for the initial move action. Please note that the numeric entries in Figure 8 are random numbers obtained from the

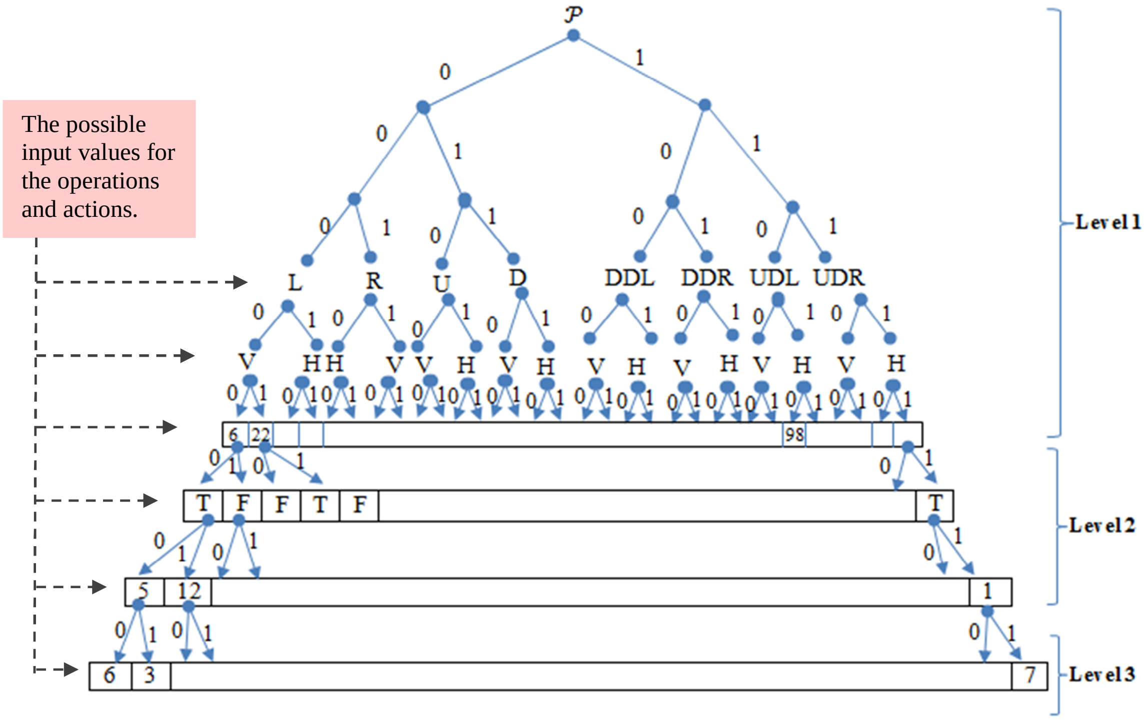


FIGURE 8. The operational knowledge

random generator. The following sections present the technical details of the encoding operations.

6.1. **Block mapping.** The block mapping uses its two actions for transforming each symbol of plaintext block into a directive. This section defines all the operations that do such a transformation.

6.1.1. *The mesh.* The mesh, as defined in [1], is a two way $N \times N$ array with horizontal and vertical dimensions. Figure 9 shows an example of the mesh. Each of the two dimensions is populated with the unicode symbols from 0 to $N-1$. The symbols in each dimension are indexed by integers $0, 1, \ldots, N-1$.

Vertical dimension

	a	b	c	u	x	d	e	t	y
y									
t									
x				C_1				C_2	
e									
c				C_3					
a									
u									
b									
d									

FIGURE 9. The mesh

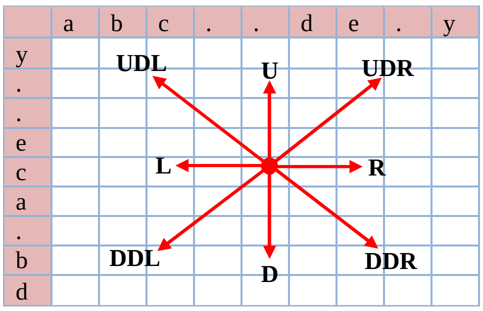
Each move from cell to another is described by two values: first, the distance of the move, which is the number of passed cells; second, the direction of the move, which designates whether the move to the lower or higher index. More specifically, we designate the move direction by the flag "+" or "−" depending whether the direction of the move is respectively to higher or lower index. For instance, moving from C_1 to C_2 has the distance of 4 since we passed 4 cells and the flag "+" since the move direction is toward a higher index.

The mesh enables us to formally define a directive. If we move within the mesh from a cell to another one by a distance x along some mapping dimension (vertical or horizontal) we designate this by "$-x$" or "$+x$" depending on whether the move is toward respectively lower or higher indexes. We call "$-x$" or "$+x$" a directive.

Mapping a symbol b_i to directive using the mesh is defined as follows. The mapping begins from some designated point (called starting point) and moves along one of the two mesh dimensions to the index of the symbol b_i in this dimension. We call the dimension that we move along the *mapping dimension*. The distance of the move and its direction with respect to the starting point is compiled as a directive "$-x$", which represents the outcome of mapping the symbol b_i. For instance, suppose we want to map the symbol "y", where the starting point is C_3 and the mapping dimension is the horizontal. We start from C_3 and move along the horizontal dimension to the index of "y" at this dimension. Since we passed 5 cells and the direction of the move is toward the higher indexes, we compile this to the directive "+5", which is the outcome of mapping "y" to the mesh.

We can reverse the mapping and obtain the original symbol from a directive if we know the starting point and the mapping dimension. The inverse mapping works as follows. Consider a directive "$\pm y$", which was produced by mapping a symbol p starting from a starting point s along a specific mapping dimension. The inverse mapping obtains the original symbol p by moving from the starting point s to lower or higher index along the specified mapping dimension. We then look up the symbol that is located in the mapping dimension and corresponds to the current index. For instance, consider the directive "+5", which was produced for the symbol "y". The inverse mapping for the directive "+5" starts from the starting point C_3 and moves 5 cells along the horizontal dimension toward the higher indexes (since the flag is "+"). We look up the corresponding symbol from the horizontal dimension, which is "y".

6.1.2. *The initial move action.* We define the initial move action by the procedure g_1: $T \longrightarrow L \times M \times D$. The input T is a set of n-place tuples $t_i < q_0, q_1, \ldots, q_n >$, where q_i is either 0 or 1. The procedure g_1 uses **Level 1** of the operational knowledge (Figure 8) to map each tuple t_i to a triple $(\psi_i, w_i, d_i) \in L \times M \times D$, where ψ_i and w_i are the two initial move parameters within the mesh and d_i is a mapping dimension. The initial move parameter ψ_i determines the direction of an initial move within the mesh. We define 8 possible initial move directions as shown in Figure 10. The second initial move parameter $w_i \in M$ is a natural number, which determines the amount of the initial move along the initial move direction ψ_i. The value $d_i \in D = \{H, V\}$ represents one of the dimensions of the mesh, where H indicates the horizontal dimension and V indicates the vertical dimension.



Instruction definition
U: move up
D: move down
L: move left
R: move right
UDL: move diagonally up and toward the left
UDR: move diagonally up and toward the right
DDR: move diagonally down and toward the right
DDL: move diagonally down and toward the left

FIGURE 10. The 8 possible initial move directions within the mesh

Referring to **Level 1** (Figure 8), the procedure g_1 works as follows. For a tuple $t_i < q_0, q_1, q_2, q_3, q_4 >$, the first three bits $q_0q_1q_2$ select one of the eight initial move directions by following the edges labeled by the three bits q_i. Continuing down the tree in the same path, the fourth bit q_3 selects the mapping dimension and the fifth bit q_4 selects the amount of the initial move. For instance, for the tuple $t_1 < 0, 0, 0, 0, 0 >$, the procedure g_1 returns the initial move direction L, the mapping dimension V, and the amount of move "6".

6.1.3. *Mesh-based mapping action.* The mesh-based mapping action takes as an input an initial starting point, a block of symbols to be mapped, and a key[1]. The mapping action proceeds as follows. The symbols of the key are transformed into a binary sequence by finding the binary equivalent to each symbol in the key. For instance, if the key is "ab", the binary representation is "01100001 01100010". For each block symbol c_i, the mapping action passes the tuple t_i to the procedure g_1 to compute 1) the initial move parameters ψ_i, w_i and 2) the mapping dimension d_i. From the current point, the mapping action first moves within the mesh boundary w_i cells along the initial move direction specified by ψ_i. If the amount of the move exceeds the boundary of the mesh, we wrap to the opposite side of the same direction and continue. Starting from this new position, the mapping action uses the designated mapping dimension $d_i \in \{H, V\}$ to map the symbol c_i as follows. The mapping action moves from the new position along the mapping dimension to the index of c_i at this dimension. The number of passed cells and the direction with respect to the position is translated into a directive $\pm x$.

6.2. **Block manipulation operation.** The manipulation operation performs two actions that alter the individual directives and modify the structure of the directive sequences.

6.2.1. *Directive masking action.* This action distorts both the distance of the directive and its flag (the sign). To do so, the directive masking action utilizes knowledge provided by **Level 2** of the operational knowledge (Figure 8). As shown in Figure 8, Level 2 has two lists of entries.

The first list has 64 entries, which are alternately populated with the symbols "T" and "F". These symbols are randomly reordered using random numbers obtained from the random generator. The reordering is performed by moving the symbol at index i to the position r_i (where r_i is a random number and $i = 1, 2, \dots, 64$). The symbols ("T" and "F") are used by the masking action as a criterion to whether flip the flag of the directive or not as follows (Flipping the flag means changing the "+" to "−" and vice versa). Given a tuple $t_i < q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7 >$, the 6th bit q_5 (along with the preceding 5 bits) is used to access one of the symbols ("T" or "F"). The symbol "T" instructs the masking action to flip the directive flag while "F" instructs the masking action to keep the flag unchanged.

The second list contains 128 entries, which are populated with random numbers obtained from our random generator. The masking action uses these random numbers to mask the distance of the directive as follows. Given the tuples $t_i < q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7 >$, the bit q_6 (along with the preceding six bits) is used to access one of the 128 entries. The accessed random number is XORed with the distance of the directive x_i (without the sign).

[1]The initial starting point is a point (x, y) within the boundary of the mesh. The integer x is the row index and the integer y is the column index. Both x and y are chosen randomly using our random generator.

6.2.2. *Directive permutation action.* The permutation action modifies the structure of the block by changing the order of its symbols. To do this, the action uses the knowledge provided by **Level 3** (Figure 8). Level 3 contains 256 entries, which are filled with random numbers obtained from the random generator. Given a tuple $t_i < q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7 >$, the permutation action uses the 8th bit (q_7) of the tuple t_i to access one of the 256 entries. The looked up value r_i is used to move the current directive $\pm x_i$ to the position r_i.

After we defined all the operations and actions of the block encoding process, we show how these actions are used to encode a block of plaintext. The encryption key is divided into 8-place binary tuples $< q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7 >$. The left 5 bits "$q_0q_1q_2q_3q_4$" are used by the Initial Move and Mesh-Based Mapping actions to produce directives for the plaintext block. The 6th and 7th bits ("q_5" and "q_6") are used by the directive masking action to mask the directives. Finally the 8th bit "q_7" is used by the permutation action to reorder the sequence of directives.

7. **Ciphered Block Decoding.**

The block decoding process is the inverse of the block encoding. It processes ciphered blocks and recovers the corresponding original plaintext blocks. Figure 11 shows the decoding process operations and actions. As the figure shows, the decoding process has roughly the same operations and actions as the encoding process except that 1) the operations and the actions are executed in reverse order and 2) some of the actions have slightly different functionality. The actions in Figure 11 whose names match those in Figure 7 have identical functionality. The rest such as Inverse Permutation have slightly different functionality and thus we will discuss them.

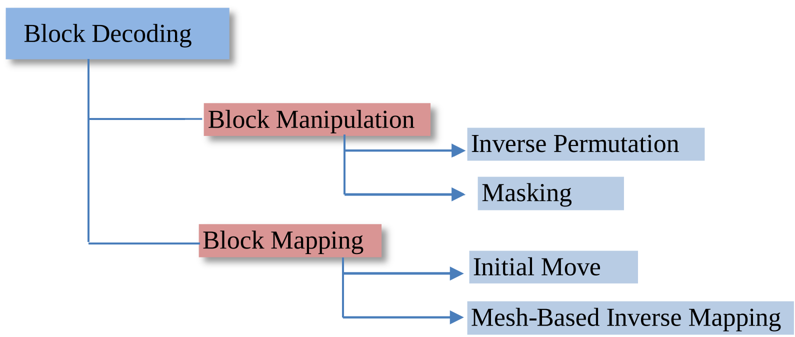


FIGURE 11. The block decoding process

Let $x_1x_2 \dots x_n$ be a sequence of directives and $t_1t_2 \dots t_n$ be a sequence of 8-place tuples, where each tuple t_i is used to produce the directive x_i. The Inverse Permutation action (which reverses the effect of Permutation action), processes a sequence of tuples backwards: from the last tuple t_n down to t_1. For each tuple t_i, the action uses t_i to access the appropriate entry in Level 3. The retrieved value from Level 3, say r_i, is used to move the directive at the position r_i (x_{r_i}) to the current position i.

Once all the directives' sequence is correctly permutated, the masking action fires. The masking action processes the sequence of directives in usual way: from left to right. In particular, the left 6 bits of the tuple t_i are used to access one of the flag masking entries of Level 2 and the left 7 bits are used to access one of the distance masking entries. The retrieved symbol from flag masking entries is used to unmask the flag and the retrieved value from distance masking entries is used to unmask the distance of the directive.

The Block Mapping operation is executed next. The output of the previous operation, say $y_1y_2 \dots y_n$, is processed in natural order from left to right. Therefore, for the tuple t_i, the *Initial Move* action determines the three parameters of the initial move in exactly the same way as in the encoding process. After the initial move action performed the initial move within the mesh, the *Mesh-Based Inverse Mapping* action uses the mapping dimension determined by the initial move action to recover the symbol encoded by the directive y_i. The inverse mapping does this by starting from the current point (determined by the initial move) and moving along the mapping dimension in a direction specified by the directive flag ("+" or "−"). The corresponding symbol in the mapping dimension is looked up, which is the original symbol encoded by the directive.

8. **Directive Substitution.** The directive substitution imposes further masking on the generated directives by replacing them with new symbols. To be effective, the new symbols should provide no information about the directives themselves. To achieve this goal, the paper proposes a substitution process that replaces directives with symbols, called instructions. These instructions specify patterns of movement within an $M \times P$ array called DIR-SUB. This array is populated with the directives from $+0$ to $+(N-1)$ and from -0 to $-(N-1)$. For instance, if $N = 256$, then DIR-SUB contains the directives from $+0$ to $+255$ and from -0 to -255 (a total of 512 directives).

We assume, without losing the generality, that $N = 256$. This means that our directives range from 0 to 255 whether flagged with "+" or "−". Accordingly, DIR-SUB can be represented by 16×32 array and is populated with the directives from -0 to -255 and from $+0$ to $+255$. The content of the array is shuffled using a sequence of random number obtained from our random generator. DIR-SUB is then used by the encoder to substitute directives with instructions and by the decoder to interpret these instructions into directives.

8.1. **The encoder process.** The encoder process works as follows. It starts always from the center of DIR-SUB (In case 16×32, the substitution process starts from the point (8, 16)). Let $\pm x$ be the directive to be substituted. First, the encoder starts from the center point and moves up or down along the column a number of positions until reaching the row that contains the directive $\pm x$. The encoder moves along the row to the left or the right a number of positions until reaching the position of the directive $\pm x$. This pattern of movement is compiled as a 9-bit instruction as follows. The leftmost bit specifies the direction of the vertical move (up or down). The following three bits specify the amount of the move up or down of the center point. The next bit specifies the direction of the horizontal move (to the left or to the right). Finally, the rightmost four bits specify the amount of the horizontal move until reaching the directive $\pm x$ within the selected row.

To further conceal the instructions, the encoding process flips bits in the generated instructions. The flipping operation proceeds as follows. For each instruction I_k, which encodes the directive $\pm x_k$, the flipping operation looks back to the rightmost digit of the previous directive $\pm x_{k-1}$. Suppose that this digit is n (0-9). The operation then flips n bits of the instruction I_k starting from the leftmost or the rightmost bit depending on the sign of the directive x_{k-1}. Since, the first instruction I_1 has no predecessor directive, the flipping operation flips the bits of this instruction using the directive in the cell $[0, 0]$ of DIR-SUB.

8.2. **The decoder process.** The decoder process takes instructions as an input and interprets them into actual directives. The decoder process first restores the state of DIR-SUB by reordering its content using the same sequence of random numbers that were used to scatter the DIR-SUB's content during the substitution (encoding). Once it is in its proper state, the decoder process starts from the center of the DIR-SUB, reads the instructions one by one, and decodes them to actual directives. It handles each 9-bit instruction as follows (see Figure 12). First, it uses the most recently decoded directive

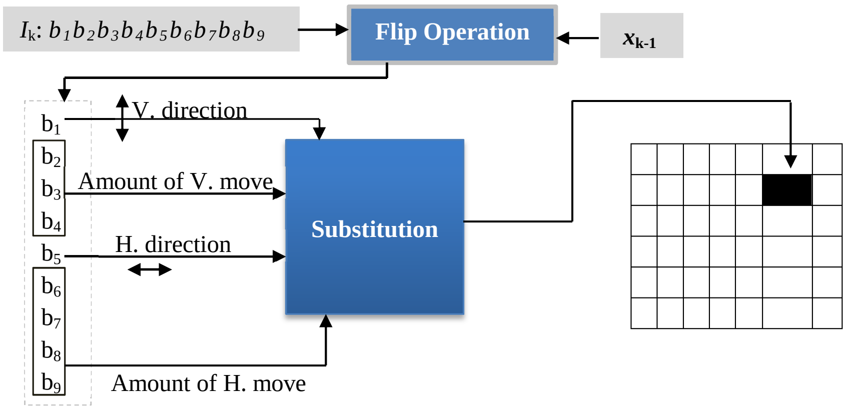


FIGURE 12. The decoder process: interpreting instructions to directives

$\pm x_{k-1}$ to flip the bits of the current instruction I_k. The bit-flipping is performed as describe above. Second, starting from the center of DIR-SUB, the leftmost bit b_1 indicates the vertical direction of the move within DIR-SUB. The next three bits $b_2b_3b_4$ enable the decoder to calculate the amount of the vertical move along the direction specified by b_1 (up/down). The next bit b_5 indicates the direction of the horizontal move within SUB. The rightmost four bits $b_6b_7b_8b_9$ are used to compute the amount of the horizontal move along the direction specified by b_5.

Before leaving this section, we point out the following. First, the output of the encoder process is a sequence of instructions, which are abstract representations for patterns of movements within DIR-SUB. As such, these instructions reveal no information about the actual directives per se due to the random scattering of the directives. Second, the instructions are the masked version of the actual instructions (due to the flipping operation).

9. **The Cipher Technique.** This section presents the proposed cipher. Namely it provides the details of the encryption process (Section 9.1) and the decryption process (Section 9.2).

9.1. **The encryption process.** The encryption process encrypts plaintext blocks $b_1b_2 \ldots b_n$ using a key $k_1k_2 \ldots k_m$. We impose no specific restrictions on the length of the block or the key. The encryption process consists of two stages: the *initialization* stage and the *encryption* stage. Figure 13 shows the main performed tasks during the initialization stage. The key expansion process extends the key into 64 bytes, which are used as a seed to the random generator. The random generator provides sequences of random numbers for performing the following tasks: 1) reorder the symbols of the mesh's two dimensions, 2) populate and reorder the entries of the binary tree in Figure 8, and 3) reorder the directive substitution table (DIR-SUB). In more details, a sequence of $2N$ random numbers is used to reorder the symbols of the vertical and horizontal dimensions (N random numbers for each). A sequence of L random numbers is used to populate and reorder the entries of the binary tree (Figure 8)[2]. Finally, a sequence of $2N$ random numbers is used to reorder the entries of the DIR-SUB table. All the symbol reordering is performed in the same way. The symbol at entry i is moved to the location r_i (r_i is random number).

[2]The sequence of L random numbers is used as follows. A sequence of 8 random numbers is used to reorder the initial move 8 directions, 16 random numbers are used to reorder the mesh's mapping dimension, 32 random numbers are used to populate the initial move amount, 64 random numbers are used to reorder the flag masking, 128 random numbers are used to populate the distance masking entries, and 256 random numbers are used to populate the directive permutation entries.

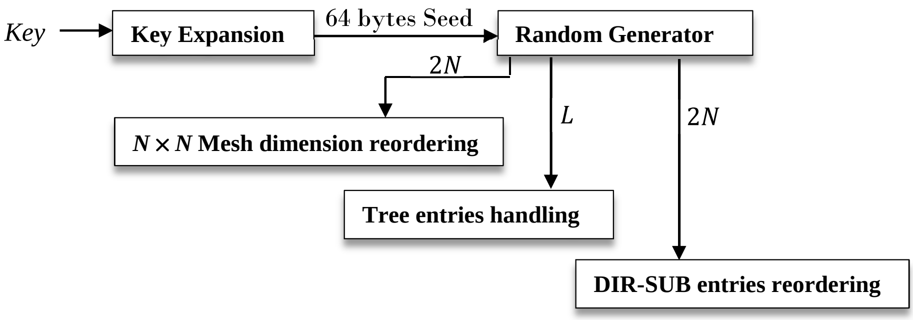


FIGURE 13. The activities of the initialization stage

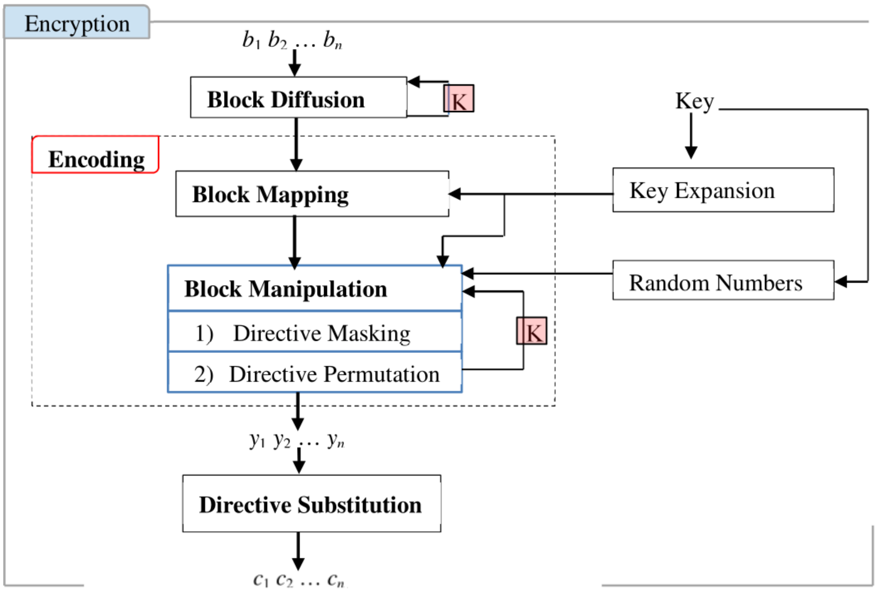


FIGURE 14. The encryption process

The encryption stage handles its input as shown in Figure 14. For each plaintext block $b_1b_2 \dots b_n$, the key expansion operation extends the key to produce the sequences $K^1, K^2, \dots, K^{k+1}$, where each K^i consists of n symbols (n is the length of the block). The **Diffusion** process handles the input block using k iterations. As previously discussed, the diffusion process ensures a high avalanche effect by amplifying change in the input block and propagating this change to impact every symbol in the output block. The output of the diffusion process is passed to the **Encoding** process, which applies two operations to the input according to the specified order: **Mesh-Based Mapping** and **Block Manipulation**. These two operations consume the key sequences K^i as shown in Figure 15 (left part). The mesh mapping uses the sequence K^1 to map the symbols of the block to a sequence of directives. The block manipulation handles its input (a sequence of directives) in k iterations each of which (iteration) consumes a key sequence K^i. In any iteration, the block manipulation applies two actions: **Directive Masking** and **Directive Permutation** – according to the order specified in Figure 14. The output of the block manipulation is a sequence of manipulated directives $y_1y_2 \dots y_n$. The Directive Substitute causes further confusions to the directives $y_1y_2 \dots y_n$ by substituting each directive with a symbol as discussed in Section 8. The outcome of this operation is the ciphered block $c_1c_2 \dots c_n$.

For every subsequent plaintext block (if any), the last sequence K^{k+1} in the extended key is used by the key expansion operation to create new sequences $K^1, K^2, \dots, K^{k+1}$ for

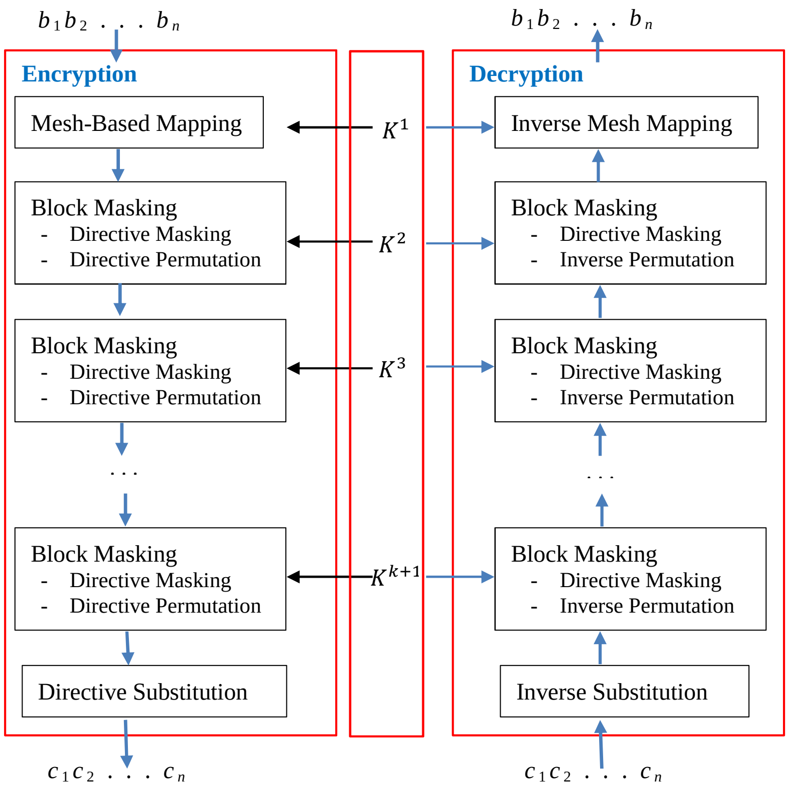


FIGURE 15. The key sequence consumption by the encryption/decryption operations

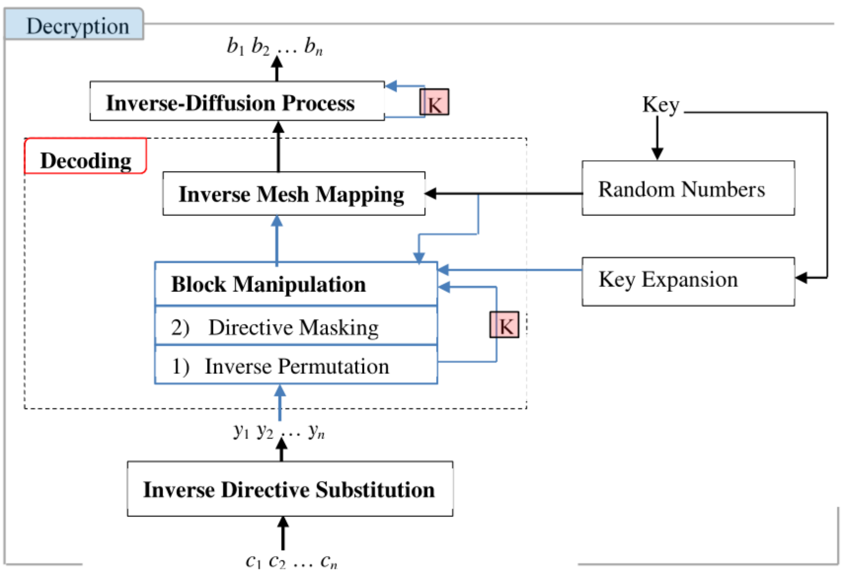


FIGURE 16. The decryption process

encrypting the new block. In this way, every plaintext block is encrypted using a new and a very different version of the key.

9.2. **The decryption process.** The decryption process takes a ciphered block $c_1c_2 \ldots c_n$ and a key as an input and returns the corresponding plaintext block $b_1b_2 \ldots b_n$ as an output. The decryption process is shown in Figure 16. Like the encryption process, the decryption process consists of two stages: the initialization stage and the decryption stage. The initialization stage is identical to that of the encryption process. It performs all the initializations in Figure 13 in addition to creating the inverse table DIR-SUB^{-1}. DIR-SUB^{-1} is used by the **Inverse Substitution** operation (Section 8).

The decryption stage is similar to the encryption stage except that it 1) uses the inverse operations and 2) executes these inverse operations in reverse order. Therefore, as Figure 16 shows, the decryption process starts from the **Inverse Directive Substitution** operation, the **Decoding** operation, and the **Inverse-Diffusion** process. Observe that the operations prefixed with the word "Inverse" reverse the effect of the corresponding operations in the encryption process. For instance, the operation **Inverse Directive Substitution** cancels the effect of the operation **Directive Substitution** and extracts the directives y_i's from the symbols c_i's.

Furthermore, the expanded key sequences $K^1, K^2, \ldots, K^{k+1}$ are used backwards (see the right part of Figure 15). Therefore, the sequence K^{k+1} is used by the **Inverse Permutation** action, the sequence K^k is used by the **Directive Masking** action, and so on.

10. **Performance Analysis.**

We evaluate our proposed technique in this section. We evaluate the randomness properties of our number generator. We then provide security analysis of the proposed technique.

10.1. **Randomness tests and basic definitions.**

Before we study the performance of the number generator and the encryption technique, we introduce some basic definitions and fundamental randomness tests.

10.1.1. *Basic definitions.*

The randomness hypotheses that we want to test are:

```
H0 (Null):  the tested data is random.
H1 (Alter): the tested data is not random.
```

Accepting H_0 or H_1 depends on a computed value called p-value and a specified value called the significance level α. The p-value is computed by the applied statistical test based on an input sequence. The significance level α is specified by the tester (e.g., 0.00001, 0.001, 0.01, 0.05 are typical values for α). In particular if p-value $\geq \alpha$, H_0 is accepted (H_1 is rejected); otherwise H_0 is rejected (H_1 is accepted). Finally, in all our tests, we assume the significance level α is 0.05.

10.1.2. *Randomness tests.*

We test the randomness properties for the output of both the random generator and encryption technique using the following randomness tests from the battery of randomness tests recommended by National Institute for Standards and Technology – NIST [16]. All the definitions were excerpted from [16].

- **Runs test**: determines whether the number of runs of ones and zeros of various lengths is as expected for a random sequence.
- **Frequency test** (**Monobit**): determines whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.
- **Discrete Fourier transform test** (*Spectral*): detects periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness.
- **Serial test**: determines if the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity in a sense that each m-bit pattern has an equal chance of appearing as every other m-bit pattern.

- **Cumulative Sums test**: determines if the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. The cumulative sums may be considered as random walks. If the sequence is random, the excursions of the random walk should be near zero.

10.2. **Key-based number generator.** We report in this section our preliminary evaluation of the number generator. We are specifically interested in studying its randomness properties. This preliminary evaluation includes a set of 19,024 keys of size 16 bytes (128 bits). We started with 128 keys each consisting of 16 identical symbols (e.g., all the symbols are zeros or a's). We then extracted 128 different keys from each key by changing only the bit i of the original key. The total number of keys obtained in this way is 16,512. Additionally, we used 2000 keys, which were generated using the online service [14]. Finally, we created 512 handcrafted keys. The total number of keys that we used in this evaluation is 19,024 different keys.

These keys are used by the number generator to produce sequences of integer numbers. Each sequence is composed of 64,000 integer numbers within the range $[0, 255]$. The generated sequences (19,024) are subjected to the randomness tests described in Section 10.1.2.

Since all the NIST's randomness tests assume binary input strings, we converted each sequence of integers to a binary string of zeros and ones. Given that all the generated numbers are integers within the range $[0, 255]$, the binary sequences can be straightforwardly obtained by finding the 8-bit equivalent of each number (e.g., the binary number equivalent to the integer 12 is 00001100).

Table 2 shows the results of the randomness tests. The table presents the randomness tests, the number of integer sequences that passed the respective test (Successes), the number of integer sequences that failed the respective tests (Failures), and the Success Rate. We set the significance level to 0.05 for all the randomness tests. This 0.05 significance level implies that, ideally, no more than 5 out of 100 integer sequences may fail the corresponding test. However, in all likelihood, any given data set will deviate from this ideal case. For a more realistic interpretation, we use a Confidence Interval (CI) for the proportion of the binary sequences that may fail a randomness test at 0.05. We therefore computed the maximum number of binary sequences that are expected to fail the corresponding test at significance level of 0.05. The maximum number is shown in the rightmost column of the table. For instance, a maximum of 1041.4 (or 1041) integer sequences are expected to fail each of the randomness tests[3].

TABLE 2. The randomness tests results for the output of the random generator

Randomness Test	Successes	Failures	Success Rate	Upper Limit of CI (0.05)
Runs test	19,024	0	100%	1041.4
Monobit test	19,024	0	100%	1041.4
Spectral test	18,362	662	96.5%	1041.4
Serial test	19,007	17	99.9%	1041.4
Cumulative Sums test	18,771	253	98.7%	1041.4

[3]The maximum number of binary sequences that are expected to fail at the level of significance α is computed using the following formula [17]: $S \cdot \left(\alpha + 3 \cdot \sqrt{\frac{\alpha(1-\alpha)}{S}}\right)$, where S is the total number of sequences and α is the level of significance.

Based on Table 2, all the sequences passed the Runs and Monobit randomness tests (Success Rate is 100%). The rates of success for the other three tests are slightly lower than 100% (these three tests are more difficult than Runs and Monobit tests). Nevertheless, all the rates of success are greatly below the maximum number of sequences expected to fail with significance level of 0.05 (See also Appendix B (external file) for more randomness testing).

10.3. **The security analysis.** The testing data were prepared according to [15]. We use –without losing the generality– the unicode symbols from 0 to 256. This enables representing the distance part of each directive by 8 bits. Testing the proposed encryption technique is conducted using the following data sets.

1) **Key Avalanche Data Set**. This data set examines the reactivity of our technique to key's changes.
2) **Plaintext Avalanche Data Set**. This data set examines the reactivity of our technique to plaintext's changes.
3) **Plaintext/Ciphertext Correlation Data Set**. This data set studies the correlation between plaintext/ciphertext pairs.

Firstly, to study the responsiveness of our technique to the key change, we created and analyzed 700 sequences of size 65,536 bits each. We used a 512-bit (64 bytes) plaintext of all zeros and 700 random keys each of size 128 bits. Each sequence was built by concatenating 128 derived blocks created as follows. Each derived block is constructed 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 modified, for $1 \leq i \leq 128$.

Secondly, to analyze the sensitivity to the plaintext change, we created and analyzed 700 sequences of size 65,536 bits each. We used 700 random plaintexts of size 512 bits and a fixed 128-bit key of all zeros. Each sequence was created by concatenating 128 derived blocks constructed as follows. Each derived block is created by XORing the ciphertext created using the 128-bit key and the 512-bit plaintext with the ciphertext created using the 128-bit key and the perturbed random 512-bit plaintext with the ith bit changed, for $1 \leq i \leq 512$.

Thirdly, to study the correlation of plaintext/ciphertext pairs, we constructed 700 sequences of size 358,400 bits per a sequence. Each sequence is created as follows. Given a random 128-bit key and 700 random plaintext blocks (the block's size is 512 bits), a binary sequence was constructed by concatenating 700 derived blocks. A derived block is created by XORing the plaintext block and its corresponding ciphertext block. Using the 700 (previously selected) plaintext blocks, the process is repeated 699 times (one time for every additional 128-bit key).

We applied the aforementioned randomness tests (Section 10.1.2) to these three data sets. We set the number of rounds K to 12 (We set K to 12 based on the results of the simulations – See Appendix A (external file)).

Tables 3, 4, and 5 show the results of the randomness tests. The tables show the randomness tests, the number of sequences that passed the respective test (Successes), the number of failed sequences (Failures), and the Success Rate. We computed in addition the maximum number of binary sequences that are expected to fail the corresponding test at significance level of 0.05. The maximum number is shown in the rightmost column of each table.

Referring to the three tables, more than 93% of the ciphertexts produced by the encryption technique passed the randomness tests. This percentage of success indicates the high performance of the proposed technique. The stability of the algorithm over the

TABLE 3. Key avalanche test

Randomness Test	Successes	Failures	Success Rate	Upper Limit of CI (0.05)
Runs test	688	12	98.2%	52.3
Monobit test	685	15	97.8%	52.3
Spectral test	663	37	94.7%	52.3
Serial test	674	26	96.3%	52.3
Cumulative Sums test	669	31	95.7%	52.3

TABLE 4. Plaintext avalanche test

Randomness Test	Successes	Failures	Success Rate	Upper Limit of CI (0.05)
Runs test	689	11	98.4%	52.3
Monobit test	684	16	97.7%	52.3
Spectral test	653	47	93.3%	52.3
Serial test	681	19	97.3%	52.3
Cumulative Sums test	673	27	96.1%	52.3

TABLE 5. Plaintext/ciphertext correlation test

Randomness Test	Successes	Failures	Success Rate	Upper Limit of CI (0.05)
Runs test	688	12	98.3%	52.3
Monobit test	682	18	97.4%	52.3
Spectral test	657	43	93.8%	52.3
Serial test	667	23	95.3%	52.3
Cumulative Sums test	675	25	96.4%	52.3

variations of the input (different keys, different plaintexts) indicates also the performance is consistent regardless of the changes in the input. Furthermore, despite the fact that some sequences failed randomness tests, the number of the failed sequences is less than the maximum expected number of sequences that may fail the tests.

11. **Conclusions and Future Work.** We described in this paper an effective encryption technique. The effectiveness of the technique is clearly reflected by the testing results (Tables 3, 4, and 5). This technique adopts effective operations that are highly vanishing the relation between the plaintext blocks input and their respective ciphertexts output. The convolutional mesh-based mapping substitutes plaintext symbols in terms of directives. The directives impose radical shifts to the plaintext symbols because they merely capture random movements in the substitution space (the mesh) and therefore bear no information about the symbols themselves. The distortion and diffusion operations add further masking to the ciphertext causing drastic increase in the diffusion, confusion, and avalanche effect. The key expansion operation allows not only expanding the key to any arbitrary length, but also ensures that the expanded versions of the key are highly different. The key-based number generator provides pseudo-random numbers. The state of the random generator periodically changes due to the different inputs (key versions) it receives as seeds, causing the encryption technique to embed greatly different random noises for each encrypted block.

Although we performed exhaustive testing for the encryption technique and the random generator, we plan to go further. First, more rigours testing may provide deeper estimations for the performance. Second, the speed of the encryption technique/random generator is vitally important. We therefore plan to conduct time requirement testing. In fact, since most of our operations manipulate their input at bit and byte level, we plan to implement the operations of the proposed technique using assembly language.

REFERENCES

[1] M. J. Al-Muhammed and R. Abu Zitar, Mesh-based encryption technique augmented with effective masking and distortion operations, in *Intelligent Computing. CompCom 2019. Advances in Intelligent Systems and Computing*, K. Arai, R. Bhatia and S. Kapoor (eds.), vol.998, pp.771-796, Cham, Springer, https://link.springer.com/chapter/10.1007/978-3-030-22868-2_54, 2019.
[2] M. J. Al-Muhammed and R. Abu Zitar, Dynamic text encryption, *International Journal of Security and Its Applications (IJSIA)*, vol.11, no.11, pp.13-30, 2017.
[3] A. Bogdanov, F. Mendel, F. Regazzoni and V. Rijmen, ALE: AES-based lightweight authenticated encryption, in *Fast Software Encryption (FSE 2013)*, S. Moriai (ed.), Berlin, Heidelberg, Springer, 2014.
[4] L. R. Knuden, Dynamic encryption, *Journal of Cyber Security and Mobility*, vol.3, pp.357-370, 2015.
[5] N. Mathur and R. Bansode, AES based text encryption using 12 rounds with dynamic key selection, *Procedia Computer Science*, vol.79, pp.1036-1043, 2016.
[6] J. Daemen and V. Rijmen, *The Design of RIJNDAEL: AES – The Advanced Encryption Standard*, Springer, Berlin, German, 2002.
[7] T. Nie and T. Zhang, A study of DES and blowfish encryption algorithm, *Proc. of IEEE Region the 10th Conference*, Singapore, 2009.
[8] M. J. AL-Muhammed and R. Abu Zitar, κ-lookback random-based text encryption technique, *Journal of King Saud University – Computer and Information Sciences*, doi: https://doi.org/10.1016/j.jksuci.2017.10.002, 2017.
[9] P. Patil, P. Narayankar, D. G. Narayan and S. M. Meena, A comprehensive evaluation of cryptographic algorithms: DES, 3DES, AES, RSA and blowfish, *Procedia Computer Science*, vol.78, pp.617-624, 2016.
[10] *NIST Special Publication 800-67 Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher Revision 1*, Gaithersburg, MD, USA, 2012.
[11] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th Edition, Pearson, 2016.
[12] R. Anderson, E. Biham and L. Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*, http://www.cl.cam.ac.uk/ rja14/Papers/serpent.pdf, Accessed in February 2018.
[13] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla and N. Zunic, *The MARS Encryption Algorithm*, IBM, 1999.
[14] *Online Random Key Generator Service*: https://randomkeygen.com, Accessed in 2018.
[15] J. J. Soto, *Randomness Testing of the AES Candidate Algorithms*, http://csrc.nist.gov/archive/aes/round1/r1-rand.pdf, Accessed in May 2018.
[16] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray and S. Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2001.
[17] J. Soto, *Randomness Testing of the Advanced Encryption Standard Candidate Algorithms*, NIST IR 6390, 1999.
[18] M. A. Ashwak, A. Faudziah and K. Ruhana, A competitive study of cryptography techniques over block cipher, *The 13th International Conference on Computer Modelling and Simulation*, Cambridge, UK, 2011.
[19] A. Juels and T. Restinpart, Honey encryption: Security beyond the brute-force bound, *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2014)*, Copenhagen, Denmark, pp.293-310, 2014.
[20] P. Bose, V. T. Hoang and S. Tessaro, Revisiting AES-GCM-SIV: Multi-user security, faster key derivation, and better bounds, in *Advances in Cryptology – EUROCRYPT 2018*, J. Nielsen and V. Rijmen (eds.), Cham, Springer, 2018.
[21] M. S. Ksasy, A. E. Takieldeen, S. M. Shohieb and A. H. Eltengy, A new advanced cryptographic algorithm system for binary codes by means of mathematical equation, *ICIC Express Letters*, vol.12, no.2, pp.117-123, 2018.

Appendices.

A. **Assessing the Impact of Number of Rounds K on the Performance of the Encryption Algorithm.** This appendix reports on our comprehensive simulations to study the impact of increasing the number of rounds K required to encrypt plaintext blocks. This parameter (K) is extremely important because increasing K greatly improves the randomness of the output (ciphertext), but it also does increase the encryption time significantly. To this end, the principal objective of this simulation is to find a golden (reasonable) value for K that guarantees both high randomness and less encryption time. In order for the simulation to result in valid estimation for the number of rounds K, the simulation must involve the use of different plaintexts and different keys – since these are the changeable inputs to the encryption algorithm. Thus, the conducted simulation herein consists of two main tasks.

- Analyze the impact of K's increase on the algorithm performance when the **plaintext is fixed**, but **keys vary**.
- Analyze the impact of K's increase on the algorithm performance when the **plaintexts vary**, but the **key is fixed**.

A.1. **Impact of K's increase: Fixed plaintext/different keys.** To analyze the impact of increasing the number of rounds K on the randomness of the output, we used the data described in Section 10.3. The number of keys is 700 random keys. The plaintext is fixed: consists of only 512 bits of all zeros. For each value of K, we created 700 sequences each of size 65,536 bits as described in Section 10.3. That is, we created 700 sequences using only one round ($K = 1$), new 700 sequences using two rounds ($K = 2$), and so on. Figure 17 shows the results in terms of the number of rounds (first column) and the count and percentage of the sequences that passed the corresponding randomness test. Figure 18 shows visually the relationship between increasing K and the percentage of the sequences that passed each of the three tests.

As the figures show, there is a clear improvement in the randomness of output as K's values increase. When the number of rounds is one ($K = 1$), a tiny percentage of the sequences passed any of the three tests. In fact, the percentage of the sequences that passed any of the tests (except for Runs test) was low when encrypting each block is

No. of Rounds K	Runs Test		Monobit Test		Spectral Test	
	No. Passed	Passed %	No. Passed	Passed %	No. Passed	Passed %
1	12	1.71%	2	0.29%	0	0.00%
2	15	2.14%	2	0.29%	0	0.00%
3	146	20.86%	6	0.86%	4	0.57%
4	292	41.71%	56	8.00%	26	3.71%
5	468	66.86%	105	15.00%	87	12.43%
6	511	73.00%	356	50.86%	198	28.29%
7	**603**	**86.14%**	**499**	**71.29%**	403	57.57%
8	**666**	**95.14%**	**588**	**84.00%**	**611**	**87.29%**
9	**674**	**96.29%**	**678**	**96.86%**	**617**	**88.14%**
10	**683**	**97.57%**	**680**	**97.14%**	**648**	**92.57%**
11	**686**	**98.00%**	**684**	**97.71%**	**659**	**94.14%**
12	**688**	**98.29%**	**685**	**97.86%**	**663**	**94.71%**
13	**688**	**98.29%**	**686**	**98.00%**	**663**	**94.71%**

FIGURE 17. The impact of increasing K on the randomness of the technique's output (Fixed plaintext, different keys)

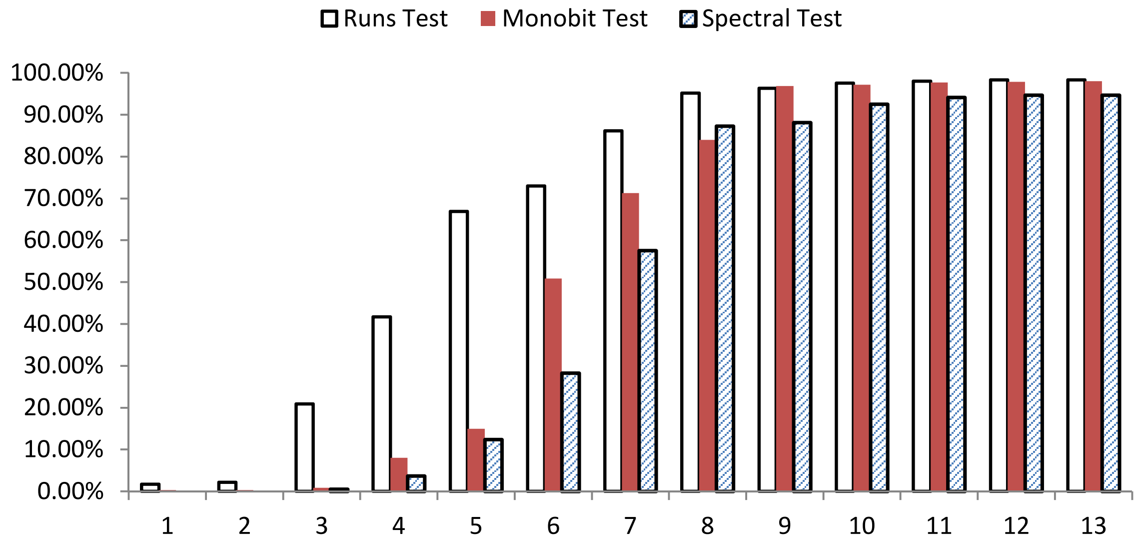


FIGURE 18. The visual representation of the data in Figure 17

performed using 7 rounds or fewer. This percentage drastically improves when we used 8 rounds or more to encrypt each block of the plaintext. Much better percentage is accomplished if $K >= 10$ for all three tests. Considering the individual randomness tests, it is obvious that the percentage of the sequences that passed the Runs test starting from 6 rounds was high while this percentage is low until the number of rounds is greater than 8 for the other two tests (Monobit and Spectral). This result can be attributed to the fact that Runs test is easier to satisfy than, say, Spectral test.

A.2. **Impact of K's increase: Fixed key/different plaintexts.** As in Section A.1, we analyze the impact of increasing K on the performance of our encryption algorithm, but at this time we fix the key and vary plaintexts. We used also 700 sequences created as described in Section 10.3: One fixed key of 128 zeros and 700 random plaintexts each of size 64 bytes (512 bits). Figure 19 shows the result of the simulations. The first column shows the number of rounds K to create each sequence. The other columns show the randomness tests and number of sequences that passed the corresponding randomness test.

Referring to Figures 19 and 20, the percentage of sequences that passed the random test are pretty close to those in Figure 17. When we use less than 8 rounds to create each sequence, low percentages of sequences pass the randomness tests. As K increases beyond 8, the percentages of the sequences that pass the randomness test drastically improve. In particular, when K is greater than 9, the percentages of passed sequences become really high regardless of the randomness test (see the boldfaced values).

We conclude our simulation by emphasizing the following important properties. First, as the number of rounds K increases so does the percentage of the sequences that pass the randomness tests. Second, using $K = 12$ rounds would be a very reasonable number since the percentage of the passed sequences does not significantly improve after this number. Third, the proposed encryption algorithm showed a consistent performance whether we fix the key and vary plaintexts or fix the plaintext and vary the keys.

B. **Further Testing for the Random Number Generator.** Nothing involving computers is ever as random as it seems. The only way to determine whether a random number generator is good enough is through careful testing. Therefore, this appendix is dedicated for conducting some important tests to measure the quality of our proposed

No. of Rounds K	Runs Test		Monobit Test		Spectral Test	
	No. Passed	Passed %	No. Passed	Passed %	No. Passed	Passed %
1	14	2.00%	4	0.57%	2	0.29%
2	17	2.43%	4	0.57%	2	0.29%
3	152	21.71%	13	1.86%	7	1.00%
4	302	43.14%	65	9.29%	24	3.43%
5	464	66.29%	111	15.86%	92	13.14%
6	516	73.71%	349	49.86%	207	29.57%
7	**599**	**85.57%**	**514**	**73.43%**	408	58.29%
8	**672**	**96.00%**	**591**	**84.43%**	**581**	**83.00%**
9	**674**	**96.29%**	**678**	**96.86%**	**617**	**88.14%**
10	**683**	**97.57%**	**679**	**97.00%**	**645**	**92.14%**
11	**686**	**98.00%**	**683**	**97.57%**	**651**	**93.00%**
12	**686**	**98.00%**	**683**	**97.57%**	**653**	**93.29%**
13	**687**	**98.14%**	**685**	**97.86%**	**653**	**93.29%**

FIGURE 19. The impact of increasing K on the randomness of the technique's output (Fixed key, different plaintexts)

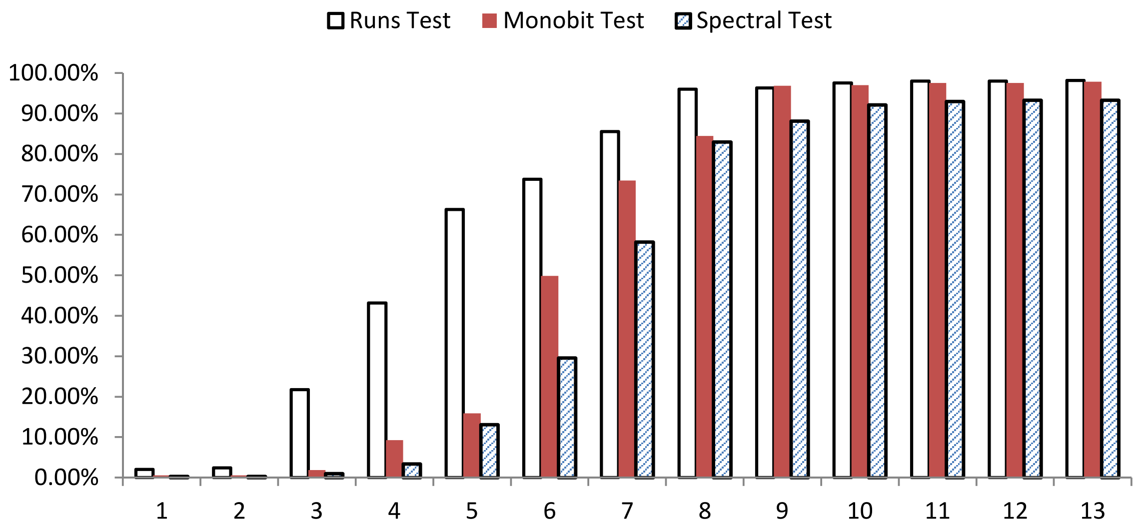


FIGURE 20. The visual representation of the data in Figure 19

random number generator and reporting the results. These tests along with the randomness tests reported in Section 10.1 are likely to give more evidences about the quality of random generator.

We conducted several important random tests in Section 10.2. This appendix provides therefore other tests to determine the quality of the random generator.

B.1. **Correlation analysis.** The correlation test checks for the presence of linear correlations between any two sequences generated using two different keys. The presence of linear correlation is not desirable because it may reveal some information about the encryption keys.

To perform this test, we created three different groups of sequences. Each group consists of 100 different sequences. These 300 sequences were randomly selected from the 19,024 sequences (Section 10.2). We then computed the correlation values between them using the MINITAB statistics software package. Table 6 shows the results of the test. Since it is difficult to report all the individual correlation values and their corresponding p-values,

TABLE 6. The correlation test results

Group	Correlation		p-value		
	Average	Max	Average	Min	Max
G1	0.004	0.07	0.52	0.334	0.990
G2	0.0025	0.0014	0.587	0.295	0.926
G3	0.0011	0.00065	0.543	0.409	0.977

the table shows only the average and max correlation value over all the possible different pairs of sequences in each category. The table shows also the p-value in terms of min, max, and average.

As the figures show, the correlation values are small, which indicate independency (no correlation) between the sequences. The p-values are all not significant since they are greater than our assumed level of significance (0.01). The high values of p-values indicate that the correlation between any pair of sequence is not statistically different from zero.

To investigate further the correlation, we conducted another test. Based on NIST testing methods, we selected 50 sequences. We created all possible different pairs of sequences (1,225 pairs). From each pair, we created a new sequence by XORing the elements of the first sequence with their respective elements from the second sequence. We then applied the Runs test on each newly created sequence to test it for randomness. Table 7 reports the results of the runs test in terms of the number of sequences passed the test and p-value (Min, Average, Max). According to the results, all the sequences passed the randomness test. The minimum p-value is 0.125, which is much higher than the assumed level of significance (0.01). This means that the tested data (1,225 pairs) show that the random generator produces independent sequences (no statistically significant correlation between these sequences).

TABLE 7. The correlation test results (NIST test)

No. Seq.	Passed Seq.	p-value		
		Min	Average	Max
1,225	1,225	0.125	0.452	0.898

B.2. **Uniformity test: Chi-Square.** The main objective of this test is to check the uniformity of the distribution of the random generator output. If the random generator is uniform, it must properly cover its range. In other words, if we divide the range into categories, there must be roughly the same number of numbers in each category for a reasonable long sequence of numbers. The importance of test is that if the random generator does not evenly cover all of its range, we will have points (or ranges) in which most of the output is accumulated. This results in security holes that can be exploited by hackers.

We randomly chose 1000 sequences out of the 19,024 sequences. To perform this test, we divided the range of our random numbers to intervals (or categories) of length 2, 4, 8, and 16. We then applied the Chi-Square for good-fitness test. We checked also that each interval should have at least 5 numbers (this is a requirement for the application of Chi-Square test). Table 8 shows the result of the test. The leftmost column shows the number of categories along with the range of the category. The rest of the columns show the average, min, and max of p-value for each category. The tested hypothesis is: the number of observed number of the random numbers in each category is not significantly

Table 8. The uniformity test results

Categories	p-value		
	Average	Min	Max
128 (category range 2)	0.413	0.201	0.876
64 (category range 4)	0.499	0.245	0.960
32 (category range 8)	0.441	0.195	0.921
16 (category range 16)	0.548	0.333	0.969

different from the expected one. In our case, the expected count of numbers is equal in all the categories.

According to the figures in the table, we accept the hypothesis since all p-values (average, min) are high (higher than 0.01), that is, the random number generator evenly covers its range.