symmetry

MDPI

Article

Strong and Efficient Cipher with Dynamic Symbol Substitution and Dynamic Noise Insertion

Ahmad A. Al-Daraiseh and Muhammed J. Al-Muhammed *

Faculty of Information Technology, American University of Madaba, Madaba 11821, Jordan
* Correspondence: m.almuhammed@aum.edu.jo

Abstract: As our dependency on the digital world increases, our private information becomes widely visible and an easy target. The digital world is never safe and is full of adversaries who are eager to invade our privacy and learn our secrets. Leveraging the great advantages of the digital world must necessarily be accompanied by effective techniques for securing our information. Although many techniques are available, the need for more effective ones is, and will remain, essential. This paper proposes a new, robust and efficient encryption technique. Our technique has an innovative computational model that makes it unique and extremely effective. This computational model offers (1) a fuzzy substitution method augmented with distortion operations that introduce deep changes to their input and (2) a key manipulation method, which produces key echoes whose relationships to the original encryption key are highly broken. These operations work synergistically to provide the highest degree of diffusion and confusion. Experiments on our proof-of-concept prototype showed that the output (cipheredtexts) of our technique passed standard security tests and is therefore highly immune against different attacks.

Keywords: encryption method; mobile-point substitution method; key echo generation; key expansion

check for updates

Citation: Al-Daraiseh, A.A.; Al-Muhammed, M.J. Strong and Efficient Cipher with Dynamic Symbol Substitution and Dynamic Noise Insertion. *Symmetry* **2022**, *14*, 2372. https://doi.org/10.3390/sym14112372

Academic Editors: Debiao He and Christos Volos

Received: 21 August 2022
Accepted: 9 October 2022
Published: 10 November 2022

1. Introduction

Although the digital world opens great opportunities to store and exchange information, it is creating real and dangerously growing challenges. In particular, threats against privacy have become critical issues that undermine our trust in the digital world. These threats stem from two perspectives. First, unless sufficiently protected, our information is highly exposed to an unfriendly environment, where adversaries are ready to exploit every opportunity to learn this valuable information. Second, as our tools for securing information reasonably advance, so do the privacy-violating tools.

Encryption is undoubtedly a key element of comprehensive information-centric security since it maximizes information protection regardless of whether the information is on a device or in a transit and provides an effective way to ward off privacy intruders. Researchers have proposed many encryption techniques. The conventional encryption techniques [1–11] offer the most widely used models whose effectiveness is built on the encryption key and the masking operations. These methods can be classified according to the way they process their input (stream or block ciphers) and according to the number of keys used (symmetric or asymmetric). The DNA techniques [12–20] offer an encryption model that makes use of the genetic properties of the DNA sequences. The general idea behind all these methods is to conceal the plaintext within complex DNA sequences. More specifically, a DNA-encoded message is first camouflaged within the enormous complexity of human genomic DNA and then further concealed by confining this message to a microdot [21]. Honey encryption techniques [22–25] provide an intriguing model for data encryption. This model purports to ensure resilience against a class of attacks called brute-force. The main idea is that for every incorrect key, the decryption process yields a plausible, but fake document. Researchers finally proposed hybrid approaches [26–29]. In hybrid approaches, two or more encryption algorithms are orchestrated and executed in a certain order. To maximize the strength of the hybrid algorithm, some of the parameters of the involved encryption methods may be tuned.

We do, however, understand that privacy intruders are working diligently to improve their hacking techniques. This incredible enhancement of hacking techniques makes the digital world the most privacy-threatening place on one hand and imposes critical challenges to the encryption techniques on the other hand. Honey encryption techniques have their own weaknesses and attacks against them are reported in [30]. Attacking techniques against the other methods do exist [31–35]. Constantly looking for highly effective techniques seems to be justifiable and supports our efforts to contend advanced threats and beat the ever-advancing cryptanalysis techniques.

This paper proposes a fully-fledged encryption technique. This technique combines a deep masking process with smart/fuzzy key manipulation operations along with a secure random generator to provide the maximum information protection. The deep masking process conceals the plaintext in enormously complex computations resulted from the fuzzy substitution and intelligent noise insertion operations. The output of the technique is further camouflaged within enormously complicated codes generated from the encryption key using a fuzzy operation.

The paper offers the following contributions. First, it proposes a deep masking process whose functionality combines both fuzzy substitutions and intelligent noise insertions. Second, it proposes an effective way to double (expand) the key. Third, it proposes an innovative method for generating key echoes, and finally it utilizes all three techniques above to provide a robust and efficient cipher.

2. Chaotic Random Number Generator

The proposed cipher uses three-dimensional Brownian motion [36]. The effectiveness of Brownian motion as a source of confusion is reported in many articles [37,38]. This motion can be simulated by the following equations.

$$\begin{aligned} x_{k+1} &= x_k + \left(Random(0,\ 1) - \tfrac{1}{2}\right) \times dt \\ y_{k+1} &= y_k + \left(Random(0,\ 1) - \tfrac{1}{2}\right) \times dt \\ z_{k+1} &= z_k + \left(Random(0,\ 1) - \tfrac{1}{2}\right) \times dt \end{aligned} \tag{1}$$

where *Random* (0, 1) returns a random value within the interval (0, 1) and $0 < dt < 1$. It is clear that the principal parameters that influence the computed values of x, y, and z using the equations above are the *seed* of the random generator and dt. To effectively link the 3D Brownian motion to our system, it is imperative to base the initialization of both the *seed* and dt on the encryption key. Suppose the key has n symbols $c1c2 \ldots cn$. The following Equations (2) and (3) use the key to compute values for these two parameters.

$$dt = Fraction\ [\sum_{i=1}^{n} c'_i \times Log_e([2^p]^{n-i})] \tag{2}$$

$$Seed = Floor\ ([\textstyle\sum_{i=1}^{n} c'_i \times Log_e([2^p]^{n-i})] \times 10^m) \tag{3}$$

where c'_i ($i = 1 \ldots n$) are the deeply transformed values that correspond to the original key symbol c_i, Log_e is the logarithm function with base e, p is the maximum number of bits that represent the used symbols (typically, $p = 8$ because we use the symbols from 0 to 255), and m is the number of decimal digits that constitute the seed. The operator *Fraction* (x) returns the fraction part of the number x and *Floor* (x) returns the largest integer less than x. It is important to mention that we calculated three different pairs for the three dimensions x, y, and z.

The paper proposes the routine Algorithm 1 for computing key-dependent values for the two parameters using the Equations (2) and (3) and *Transform* (.) subroutine (Figure 1).

The subroutine *Transform* (Figure 1) plays a vital role in deeply manipulating the key (n symbols) and producing a new sequence with n symbols but with a higher entropy. It uses a random behavior (subroutine *Randomize* ()) and an XOR operation to produce values pc's that result from XORing the input key symbol c' with p bits extracted from the rightmost of L using the division remainder operator (Mod). Observe, the variable L accumulates the effect of all the previously processed symbols due to the way it is updated. This effectively enables the previous symbols to influence the transformation of the symbol being processed. Additionally, thanks to the randomization of L, the influence of the previously processed symbols induces high confusion, making the transformation produce very different sequences.

Algorithm 1 Computing values for *Seed* and *dt*

Key: $C_1C_2 \ldots C_n$
Key′ = ($C'_1\ C'_2 \ldots\ C'_n$) = *Transform* (Key)
Compute *dt* using Equation (2)
$C'_1\ C'_2 \ldots\ C'_n$ = *Transform* (**Key′**)
Compute *Seed* using Equation (3)

$Transform(k_1 \ldots k_n)$ /**original input key symbols*/
$L = k_1k_2k_3k_4$ /** the variable L is the concatenation of the leftmost 4 symbols of the key*/
$L = Randomize(L)$ ⟶
For i=1 to n **Do**
  $pc = k_i \oplus Mod(L, 2^p)$
  $Res = Res + Char(pc)$
  $L = L \oplus k_i$
  L = Randomize(L)
Res = $k'_1k'_2 \ldots k'_n$
For i=n downto 1 **Do**
  $pc = k'_i \oplus Mod(L, 2^p)$
  $Seq = Char(pc) + Seq$
  $L = L \oplus k'_i$
  $L = Randomize(L)$ ↗
Return Seq
	$Randomize(L)$
	  $L = L \oplus (L << 13)$
	  $L = L \oplus (L >> 17)$
	  $L = L \oplus (L << 5)$
	  Return L

Figure 1. The transformation subroutine (arrows just indicators of call direction).

After initializing the parameters, the chaotic random numbers (x_r, y_r, z_r) are generated using the following algorithmic steps. As a convention throughout the paper, we call the sequence of random numbers x_r's the X-channel. Likewise, we call the sequences of random numbers y_r's and z_r's the Y-channel and Z-channel, respectively.

Generating chaotic numbers (x_r, y_r, z_r)
Repeat
Execute the Equation (1)
x_r = Mod (floor ($x_{k+1} \times 10^{14}$), 2^p)
y_r = Mod (floor ($y_{k+1} \times 10^{14}$), 2^p)
z_r = Mod (floor ($z_{k+1} \times 10^{14}$), 2^p)
End

3. The Deep Masking Round

Secure ciphering requires effective transformation of the symbols from their plaintext space to an entirely different space in which the relation between the plaintext symbols and the resulting symbols is untraceable. The deep masking round consists of two effective methods (the substitution method and the distortion method) that deeply mask the blocks of plaintext. Each method uses different techniques to make sharp changes to its input block. Their collective impact on the input results in an output block whose relation to the input block is highly complicated. This section therefore first discusses these two methods (Sections 3.1 and 3.2) and then discusses how these two methods work synergistically to perform deep masking for its input blocks (Section 3.3).

3.1. The Substitution Method

The substitution uses a dynamic method to replace the plaintext symbols bi with new ones ci. It adopts a fuzzy and data-dependent computational model whose functional behavior depends not only on the symbols to be substituted, but also on move operations that fuzzify the substitution operation by executing different move patterns within the substitution space. This section explains the main constituents of the substitution method: the substitution space, the move operations, and then concludes the section by offering a specific way for selecting a particular move operation.

3.1.1. The Substitution Space

The substitution space (M-TAB) is a $2^{p/2} \times 2^{p/2}$ array that contains all possible permutations of p bits. The entries are initially organized in M-TAB as specified by AES encryption technique. The substitution technique uses M-TAB for mapping symbols to new ones. Specifically, the substitution method maps a symbol bi by splitting its bits into two halves, where the left half indexes a row, and the right half indexes a column. The indexed cell is the mapping outcome. For instance, to map the symbol "i" ("01101001"), the left four bits ("0110") index row 6 and the right four bits ("1001") index column 9. The value at the cell (6, 9) is, therefore, the outcome of mapping the symbol "i".

3.1.2. The Move Operations

The move operations add a high degree of fuzziness to the substitution. Table 1 shows the move operations along with descriptions of their functionality. The operations execute either unidirectional or bidirectional moves. Left (g) and Right (g) are examples of unidirectional move operations. The former moves g steps to the left of a specific cell (r, c), while the latter moves g steps to the right of the cell (r, c). For instance, the operation LU (5, 12) performs moves in two different directions: move 5 cells to the left then 12 cells up. LU (i, g) (i, g =1 ... $2^{p/2}$) is an example of a bidirectional operation since it executes a move with two different directions: it first moves i positions to the left of a cell (r, c) and then moves g positions up. It is worth mentioning that we use (i + r) mod $2^{p/2}$ instead of simply adding i to r, to stay in the grid. Similarly, we use c and g.

Table 1. The move operations.

Operation	Functionality	Operation	Functionality
Left (g)	Move g positions from the current position tothe left	*Right* (g)	Move g positions from the current position to the right.
Top (g)	Move g positions from the current position tothe top	*Down* (g)	Move g positions from the current position to the down
LU (i, g)	Move first left i steps, then move up g steps	*LD* (i, g)	Move first left i steps, then move down g steps.
RU (i, g)	Move first right i steps, then move up g steps	*RD* (i, g)	Move right left i steps, then move down g steps

Each move operation has an inverse operation. Table 2 shows the move operations and their respective inverse operations. For instance, if *Left* (g) performs a move within M–TAB from the position (r, c) to (r', c'), *Right* (g) performs a move back from (r', c') to the original position (r, c). In addition, if we move from position (r, c) to the new position (r', c') using the operation *LU* (i, g), we can move back from the new position (r', c') to the original position by executing the inverse operation *RD* (i, g).

Table 2. The move operations and their inverse.

Operation	**Its Inverse**	**Operation**	**Its Inverse**
Left (g)	*Right* (g)	*LU* (i, g)	*RD* (i, g)
Up (g)	*Down* (g)	*LD* (i, g)	*RU* (i, g)
Right (g)	*Left* (g)	*RU* (i, g)	*LD* (i, g)
Down (g)	*Up* (g)	*RD* (i, g)	*LU* (i, g)

3.1.3. Move Operation Selection

The way in which the move operations are selected is extremely important. Chaotic but key-based selection enables the substitution method to execute haphazard patterns of moves within the substitution space. These move-patterns, though haphazard, are reproducible due to their dependency on the key. Such a functional behavior induces substantial confusion in the substitution process and yields a large shift from the plaintext space.

The paper proposes a selection method that uses key-driven chaotic numbers (from X, Y and Z-channels) to chaotically choose a move operation. Figure 2 outlines the selection process. The selection method makes use of three lists (OP, i, g), each with 2^p entries. The list OP is populated with even replications of the move operations (Table 1). The lists i and g provide values for the arguments of the move operations and are populated with the integers in the range $[0 \ldots 2^{p/2}]$. The entries of each list are randomly reordered using a sequence of random numbers obtained from the chaotic system.

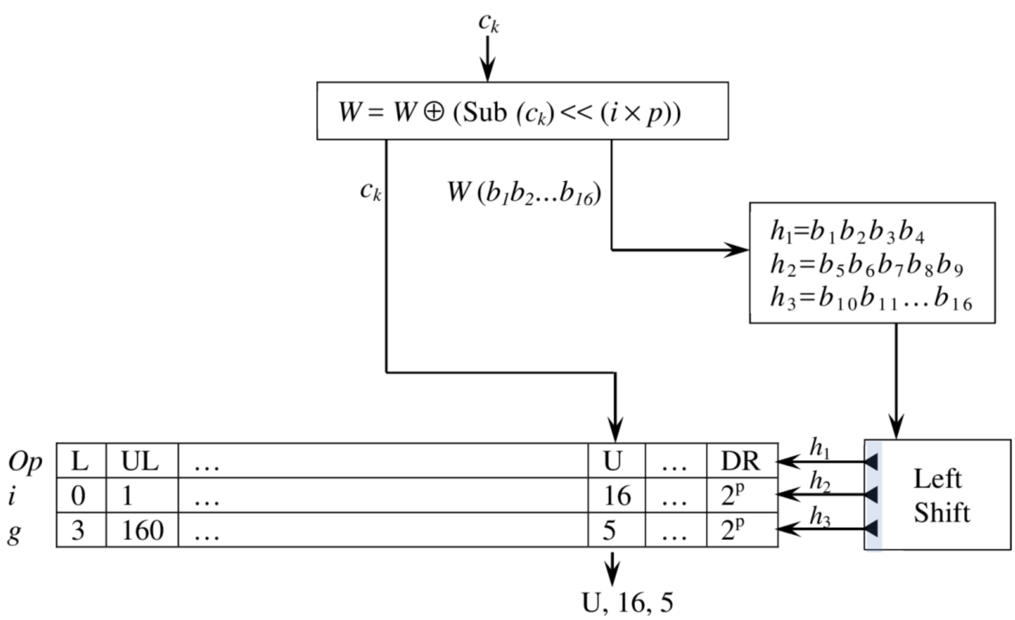

Figure 2. The move operation selection method.

The chaotic selection receives the chaotic input value c_k (obtained from X, Y, *or* Z-channel in a round robin fashion) and this input c_k is consumed by two actions that both stimulate the chaotic behavior in the selection method. The first action left shifts the entries of the three lists (OP, i, g) before indexing them. The action maintains for this purpose a

16-bit state variable W, which is initialized with a chaotic value from the chaotic system and is updated using the formula: $W = W \oplus (c_k << (i \times p))$, where i is the rightmost bit of c_k and p is the maximum number of bits that represent a symbol. The 16 bits (b_i's) of the state variable W are split to three values h_1, h_2, and h_3 and these values are used to left-shift the three lists as shown in Figure 2. The second action uses the chaotic input symbol c_k to index the same entry of the three lists (OP, i, g) to retrieve a move operation from OP and two values for the arguments of the selected method from the lists i and g. If the move operation is unidirectional, the extra value from the list i is discarded.

3.2. Symbol-Distortion Method

This method manipulates the output symbols of the substitution method. The method defines several distortion operations and proposes a specific way of selecting any of them. The distortion method processes its input symbols using either individual operations or composite operations, where each composite operation consists of two or three all-different operations that execute sequentially left to right.

3.2.1. Distortion Operations

Table 3 presents three operations that manipulate the individual input symbol. All these operations perform bitwise processing on the input symbol b. *Mutate* (b, u) XORes the input symbol b with a selected pattern u, where u is composed of 0′s and 1′s. Unlike *Mutate* (.), *Swap* (b, u) operation modifies the internal structure of the input symbol but not its bits. In particular, this operation permutes the bits of b according to some pattern u, where the symbol b consists of p bits and the pattern u consists of p integers $d_1 d_2 \ldots d_p$, where $d_i \in [0 \ldots p]$. The swap operation permutes the bits of b by swapping $b[i]$ with $b[d_i]$ only if $d_i > 0$. The value d_i =0 instructs the swap operation to skip to the next digit in the pattern without permuting. The *L-Rotate* (b, n) operation rotates the bits of the input symbol n positions to the left. The number of positions n is an integer less than the number of bits representing the symbol b. The selection of the patterns and n is explained below.

Table 3. Noise operations.

Operation	Functionality
Mutate (b, p)	Flips a number of bits of the input symbol b based on the input p. The pattern p specifies the positions of the bits to be flipped. For instance, the pattern “01001010” flips the 2nd, 5th, and 7th bits. The flipping is performed by XOR operation: $b \oplus p$. Note if the symbol b is represented by 8 bits, then we have 256 possible mutation operations
Swap (b, p)	Swaps bits from b based on the pattern p. For instance, if the pattern “05020000”, repositions the 5th in the second position, the 2nd in the 4th position.
L-Shift (b, n)	Left shifts the bits of b by n bits.

There is a distortion operation inverse for each distortion operation. When a symbol x is processed using *Mutate* (x, e), it can be restored by executing this function with the same pattern e. When a symbol x is processed using *Swap* (x, z) to yield the symbol y, the symbol x can be restored using the inverse operation $Swap^{-1}$ (y, z). The functionality of $Swap^{-1}$ (y, z) is slightly different from that of *Swap* (y, z): it parses the swapping pattern z from right to left and swaps the bits when $d_i > 0$ by swapping $b[d_i]$ with $b[i]$. When a symbol x is processed using *L-Rotate* operation, it can be restored using the *R-Rotate* (right shift) operation. Finally, when a symbol x is processed using a composite operation, the symbol x can be restored by executing the composite inverse operation in the reverse order. For instance, if x is processed by executing the composite operation *Swap* () → *Mutate* () → *L-Rotate* (), the symbol can be restored by executing the composite operation in the reverse order *R-Rotate* () → *Mutate* () → $Swap^{-1}$ ().

To effectively use the distortion operations, we need to define (1) the flipping and swapping patterns and (2) the way in which the noise operations are selected to process a specific input. The patterns used for flipping bits are the integers from 0 to $2^p - 1$ and we do not discuss them any further. However, the swapping patterns and the noise operation selection method need further discussion.

3.2.2. Swapping Pattern Generation Process

The distortion operation *Swap* requires swapping patterns for manipulating the bits of its input symbols. Effective generation of the swapping patterns must depend on the encryption key so that different keys produce different swapping patterns. Let us suppose that each symbol is represented by p bits. A swapping pattern consists of p integers d_i, where $d_i \in [0 \ldots p]$. For instance, if each symbol is represented by eight bits, the swapping pattern consists of eight integers in the range $[0 \ldots 7]$. Figure 3 outlines the three stages for creating swapping patterns along with their logic. The initialization stage populates the list *SWL* with $N = 2^p$ integers, each integer is in $[0 \ldots p]$. The population stage receives two input values: $\alpha \in (0, 1]$ is the percentage of the non-zero digits in the list *SWL* and p is the number of bits that represent a symbol—we call α the intensity of active swapping. To give each non-zero digit that same chance to be part of a pattern, we set the number of non–zeros digits in the list *SWL* to $M = MULT^p\ (\alpha * N)$, where $MULT^p(x)$ returns the least upper bound of its argument x that is divisible by p. In this case, each non-zero digit is repeated M/p times in the list *SWL*. The remaining "$2^p - M$" entries in the list *SWL* are zeros. After the list *SWL* is fully populated, its entries are randomly shuffled using a sequence of random numbers $r_1 r_2 \ldots r_N$. The entries of *SWL* are randomly shuffled by swapping the entry at index i with the entry at index r_i.

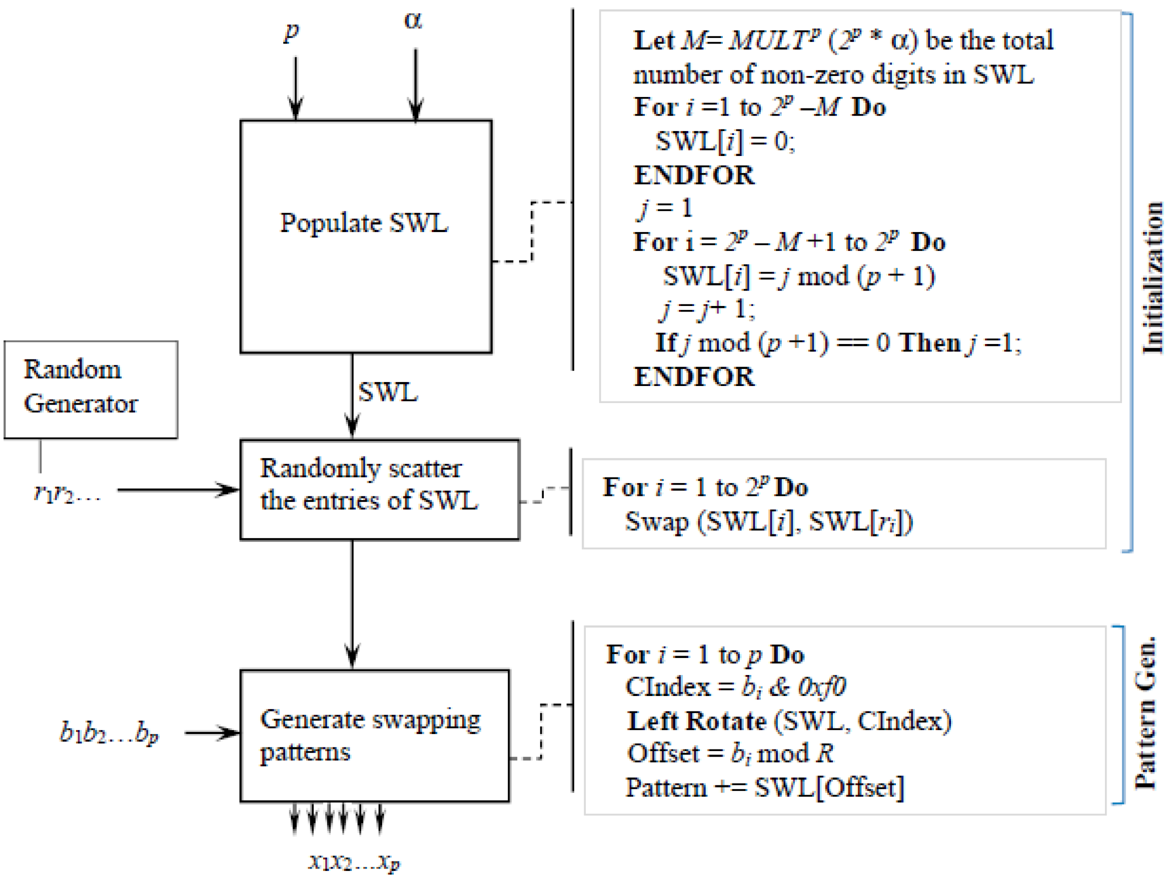

Figure 3. The swapping pattern generation operations.

To illustrate, suppose that $p = 8$ and $\alpha = 0.65$. The total number of non–zero elements in the list *SWL* is 0.65 * 256 = 166.5. The operation $MULT^p$ (166.5) returns the least upper bound divisible by 8, which is 168. Therefore, the list *SWL* is populated with 168 non-zero digits, where each of the digits 1 . . . 8 is repeated 168/8= 21 times. The remaining entries (256 – 168 = 88) are populated with zeros.

Once the *SWL* is fully initialized, the pattern generation operation starts. The input to this method is a sequence of p symbols $b_1 b_2 \ldots b_p$. (The sequence of the symbols b1b2 . . . bp is obtained from the key in a process (key doubling) that is described in later sections.) The output is a swapping pattern with p digits, each of which is an integer in $[0 \ldots p]$. The output is a swapping pattern with p digits each of which is an integer in $[0 \ldots p]$. The method uses an effective computational model that uses each symbol b_i to obtain two integers $CIndex = b_i/L$ and $Offset = b_i$ Mod R, where $L = R = 2^{p/2}$ and p is the number of bits that represent the symbol b_k. The number *CIndex* represents how many positions the SWL is left rotated. After the left rotation is performed, the entry at the index *Offset* is retrieved and appended to the intermediate pattern. The operation "Generate swapping patterns" can repeat with new input until the desired number of patterns is created.

3.2.3. Distortion Operation Selection

After discussing the distortion operations, we propose a specific method, called the symbol-sensitive sliding technique, of selecting any of them to manipulate the input symbols. Figure 4 shows the main components of the sliding technique and its logic. It utilizes a 4×2^p array, a ring of 2^p entries, and a state variable G^i (initialized to 0). Each of the entries of the first row contains either a single distortion operation ⟨*Mutation* (*M*), *Swap* (*S*), *L-Shift* (*L*)⟩ or a composite operation consisting of two or three operations. The composite operations are: *LS*, *SL*, *LM*, *ML*, *SM*, *MS*, *LMS*, *LSM*, *MLS*, *MSL*, *SLM*, and *SML*. These operations (single or composite) are replicated evenly in the array. If there are fewer entries than the number of the distortion operations, these entries are populated with *N* (NULL operation that does nothing). The second row contains the swapping patterns W_i ($i = 1 \ldots 2^p$). The third row contains mutation patterns M_i, which are the integers from 0 to $\langle 2^p - 1\rangle$. The fourth row contains the amount of left shifting (L_i), which are the integers $L_i = i$ Mod p ($i = 1 \ldots 2^p - 1$). These four rows are randomly scattered using a sequence of random numbers from *X* to channel. The 2^p entries of the ring contain random numbers, where each entry contains a number from the range [0, 1].

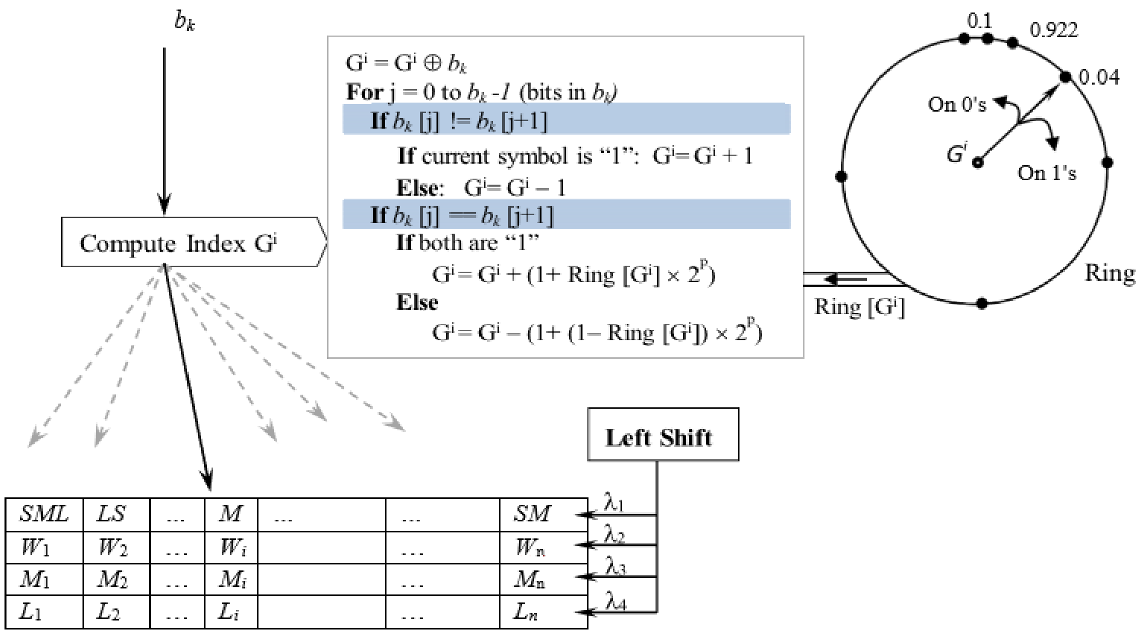

Figure 4. The noise operation selection method.

The sliding technique takes a symbol b_k as input. It uses its logic to generate a new value for the state variable G^i and use this variable to access one of the four-row arrays columns. To update G^i, the sliding technique exploits both the value of the input symbol b_k (b_k is a key symbol) and its bit structure. First, G^i is XORed with the current input symbol b_k. The new value of G^i is additionally updated using the structure of b_k's bits. In particular, the sliding technique parses the bits of the input b_k one at a time and tunes the value of G^i as follows. The pointer G^i moves around the ring clockwise or counterclockwise depending on whether the currently parsed bit is respectively "1" or "0". The amount of the move is fully determined by the 1-lookahead bit (the bit immediately after the current bit). If the 1-lookahead bit is different from the current bit, the pointer G^i moves clockwise or counterclockwise by only one. If the 1-lookahead bit is the same as the current bit, G^i moves clockwise or counterclockwise by 1 + e. The value e is called the sliding value and is computed by multiplying the value γ (from the ring) to which G^i currently points and 2^p(the total number of the ring's entries). For instance, if G^i currently points to the value 0.0456 and the number of entries in the ring is 256, G^i is moved from its current position by "1 + 0.0456 * 256 = 12" steps clockwise/counterclockwise based on whether the current and lookahead bits are $1'$s or $0'$s. After processing all the bits of b_k, the pointer G^i indexes one of the columns, where the corresponding distortion operation along with the necessary arguments for this operation are accessed. For instance, if the corresponding operation is S (swapping), only the swapping pattern is accessed.

Prior to processing any new input symbol $b_{k+1,}$ the array's rows are left shifted using the value to which the pointer G^i is currently pointing. For instance, if G^i is currently pointing to value γ $(0.\lambda_1\lambda_2 \ldots \lambda_m) \in (0, 1)$, the rows 1 through 4 are left shifted by λ_i (i = 1, 2, 3, 4). Note due to the specific way in which G^i is calculated, changes in any input symbol will affect all the subsequent values of G^i. Furthermore, due to the left shifting after processing each input symbol, the indexing outcome changes for every subsequent input symbol, making the distortion operation selection highly fuzzy.

3.3. The Deep Masking Process

Referring to Figure 5, the deep masking process consists of two operations: the substitution operation $\langle 1, 2, 3\rangle$ and the distortion operation $\langle 4, 5\rangle$. (The numbers at the top of the boxes represent the execution order.) The substitution operation replaces the symbols $b_1b_2 \ldots b_n$ of plaintext using the actions (1, 2, and 3). The Move Operation Selection uses the symbol x to choose one of the move operations M -Op_l and its arguments as described in Section 3.1.3. The input plaintext symbol b_k designates a location (i, j) within *M-TAB*, where i is the left half bits of b_k and j is its right half bits. The selected move operation M -Op_l is then executed starting from (i, j) to yield a new location (L_1, L_2) within the substitution space. The symbol T is retrieved from the new location (L_1, L_2) of the substitution space *M-TAB* as an intermediate substitute for the input plaintext symbol b_k. To increase the fuzziness of the masking process, the symbol x is also used to obtain a distortion operation D-Op_z along with the required arguments as described in Section 3.2.3. The selected distortion operation is executed on the symbol T to yield the new symbol T', which is the deeply masked substitute for the input symbol b_k.

3.4. Deep Masking Inverse Process

The deep masking inverse process reverses the effects of the deep masking process. That is, this inverse process restores the plaintext symbols from the masked symbols. Figure 6 shows the operations of the deep masking inverse. The logic of this process is similar to that of the deep masking process except that the order of the execution is reversed: distortion operation inverse is executed first then the substitution operation inverse. The distortion operation inverse receives the masked symbol T' as an input and removes the masking effect of the distortion operation (used during the masking). It removes the impact of the distortion operation by using the input x to retrieve the appropriate distortion operation inverse and executes this operation on the input symbol T'. The output is a new symbol T that is passed to the substitution operation inverse for removing the impact of the substitution operation as follows. The symbol T is looked up from the M-TAB and its location within the substitution space (L_1, L_2) is passed to action 5 for further processing. Next, the deep masking inverse process uses the symbol x to select a move operation inverse M-Op^{-1}. The selected move inverse operation slides back from the index (L_1, L_2) to the original index (i, j). Finally, the bits i and j are concatenated (ij) to yield the original symbol b_k.

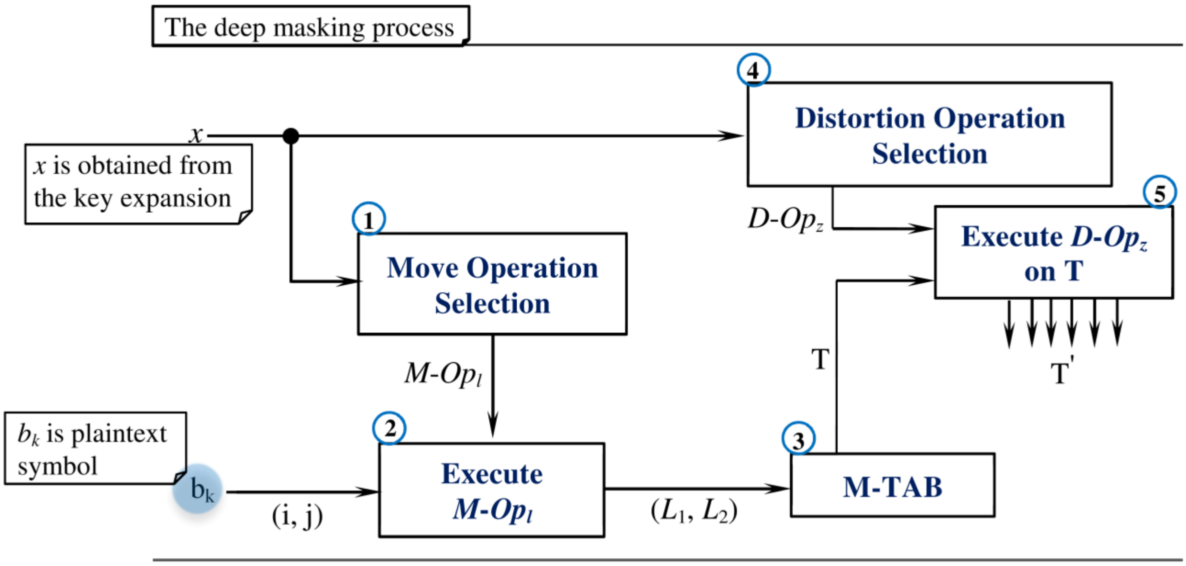

Figure 5. The deep masking process.

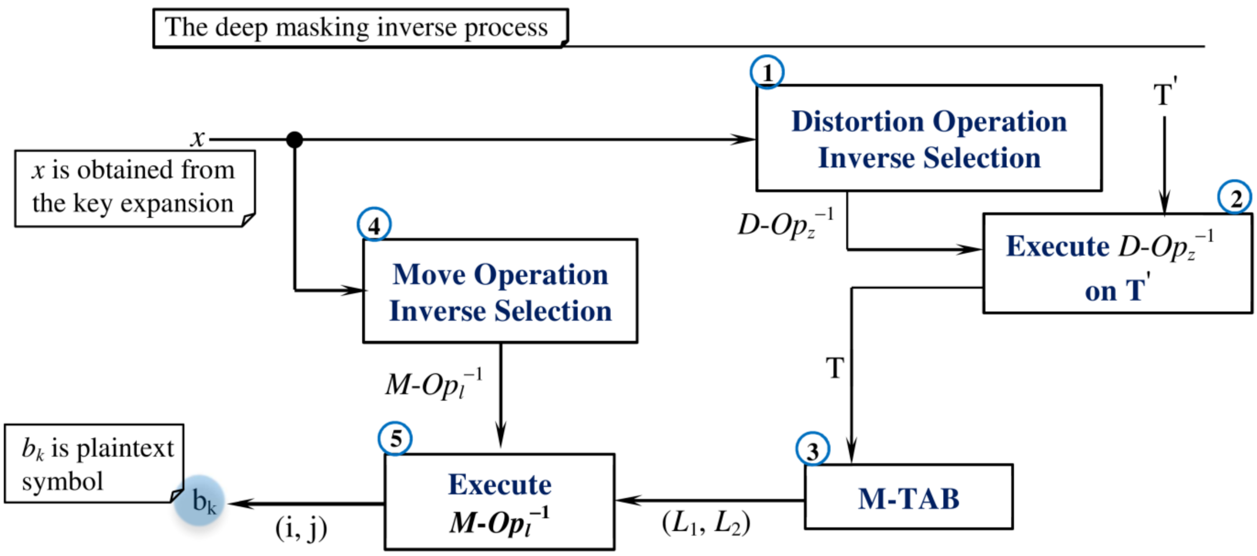

Figure 6. Deep masking inverse process.

4. Key Doubling Operation

The Key Doubling operation expands its n-symbol sequence input to a $2n$-symbol sequence. It is intended mainly to extend the key. Figure 7 shows the main four actions of this operation.

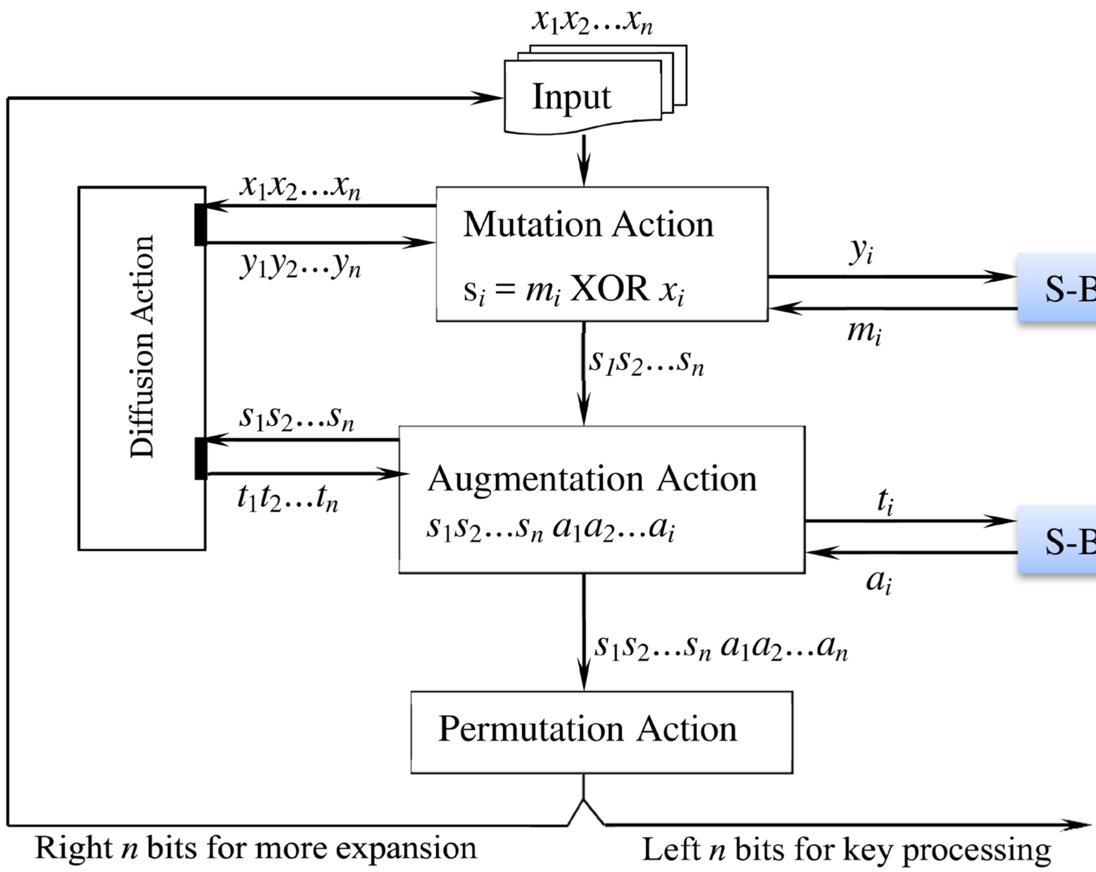

Figure 7. The Key Doubling operation algorithmic steps.

4.1. Diffusion Action (D-Action)

The diffusion action processes its input in two passes: Forward-pass and Backward-pass (See Figure 8). Due to its bidirectional processing model, the diffusion action (1) is highly sensitive to the input changes regardless of their position and magnitude and (2) makes any change that occurs to a symbol in its input affect all the symbols in the input sequence. The forward-pass uses M-TAB to substitute the input symbol b_1 to yield a new symbol c_1. For the remaining input symbols b_i ($i > 1$), the forward-pass first XORes b_i with the previous output symbol c_{i-1} and substitutes the outcome of the XOR to produce the symbol c_i.

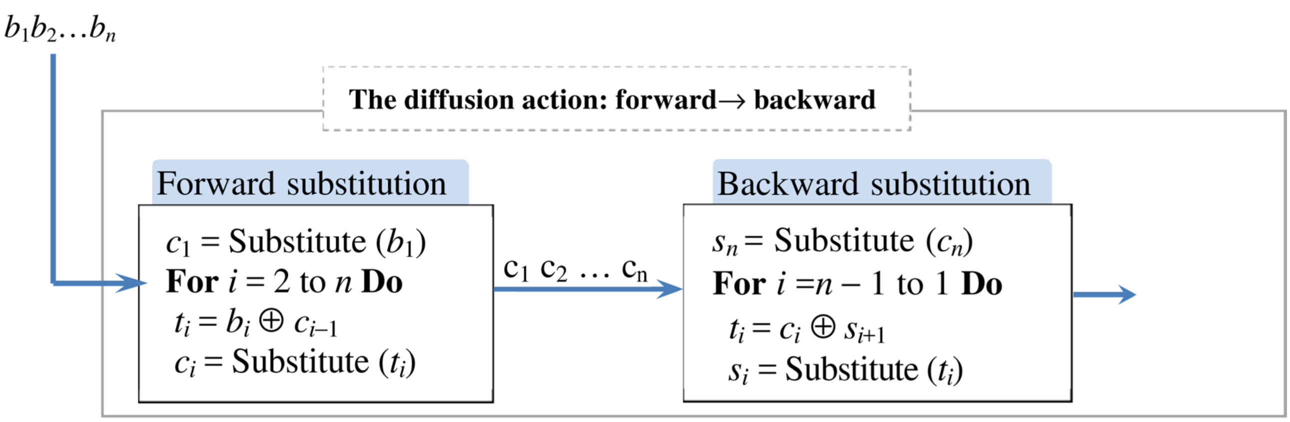

Figure 8. The algorithmic steps of the diffusion action.

The backward-pass processes the output of the forward-pass to deepen the mutual-impact between the symbols. It uses similar logic as the forward-pass except that it processes the input backward: from the end of the input block. The backward-pass therefore substitutes c_n to yield the output symbol S_n. For the remaining symbols c_i ($i = n$–1, n–2, . . . , 1), it performs an XOR operation between the current input symbol c_i and the previous output symbol S_{i+1} and substitutes the outcome of the XOR to yield the output symbol S_i.

The forward-pass drives the impact of the symbol b_i forward to influence the subsequent symbols b_j ($j > i$). The backward-pass drives the impact of the symbol b_k backward to affect the predecessor symbols b_i ($i < k$). Thanks to the dual-direction processing, the diffusion action is highly sensitive to symbol-changes and intensifies the mutual-influence between input symbols.

4.2. Permutation Action (P–Action)

The permutation action adopts a data-dependent functionality to scramble the order of the symbols of the input sequence. Algorithm 2 delineates the processing steps. In such a data-dependent functionality, the symbol x_i determines the new position for the immediate successor symbol x_{i+1} (within the input sequence). As Algorithm 2 shows, the permutation action moves the symbol x_2 to the new position determined by the ascii index of the symbol x_1. When processing the remaining symbols, the data-dependence is intensified even more by introducing other factors. For $i > 1$, the new position of the symbol x_{i+1} depends not only on its immediate predecessor symbol x_i but also on the last point of insertion *LIP*. When the ascii index of the symbol x_i is greater than the value *LIP*, the permutation action moves x_{i+1} to the new position determined by the formula: $x_i \oplus LIP$. If otherwise, the permutation action moves x_{i+i} to the new position determined by the ascii index of $x^{\sim}$ (complement of x_i).

Algorithm 2 The algorithmic steps of the permutation action

PERMUTE (x_i, x_{i+1})
    If $i = 1$, the symbol x_{i+1} is moved to the position $k = x_i$
    For all $i > 1$, PERMUTE (x_i, x_{i+1}) moves x_{i+1} to a new position as follows

a. Compute the position k

    If $LIP > x_i$ move x_{i+1} to the position $k = x_i$ *XOR LIP*
    Else move x_{i+1} to the position k = *Complement* (x_i)

b. Update $LIP = k$

4.3. Mutation/Augmentation Actions

The mutation action utilizes both the diffusion action and the M-TAB substitution to impose radical changes to the original input sequence $x_1x_2 \ldots x_n$ and decays the relationship between the input and the output $y_1y_2 \ldots y_n$. In particular, the diffusion action makes sharp changes to its input making the output far different from the input. Furthermore, the M-TAB substitution also has the impact of shifting the input sequence to a different space (set of symbols) that dissolves the correlation to the original symbol (As reported in [39], substituting the symbols of an input sequence using the M-TAB deteriorates the relationship between the sequence input and the output of the substitution). Collectively, the final output sequence $s_1s_2 \ldots s_n$, which is the outcome of the XOR operation between the original input sequence and the processed sequence, has no correlation to the original input. The augmentation action essentially carries out the same steps, except that the outcome of the substitution a_i is appended to the end of the input $s_1s_2 \ldots s_n$.

Using these four actions, the input doubling operation is executed as follows. The input $x_1x_2 \ldots x_n$ is deeply manipulated using the mutation action. The augmentation action doubles the input to produce a sequence of 2n symbols. Finally, the permutation action scrambles the block by imposing data-dependent reordering. The right n symbols are passed back to produce more $2n$-symbol sequences. The left n symbols are used to support different operations of the encryption/decryption process.

5. Key Echo Generation Method

This method is a three-stage process that uses the encryption key to produce a long stream of codes for hiding the ciphertext symbols. The method conservatively passes the key through sophisticated processing operations that dissolve the trace of the key within the enormously complicated generated codes. Figure 9 shows the logic of the processing stages.

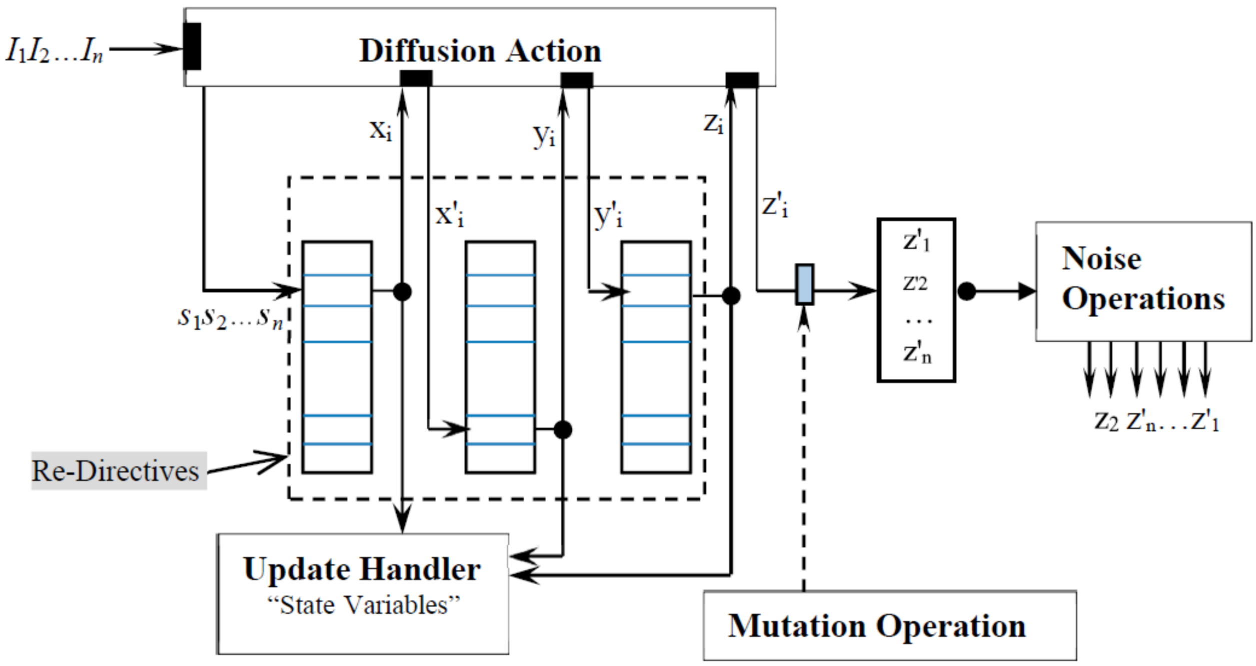

Figure 9. The key echo generation operation.

The first processing stage consists of the Diffusion Action and the Re-Directives. The diffusion action uses D-Action (Section 4.1) to maximize the avalanche effect due to the input changes.

The Re-Directives operation is a multi-stage mapping operation that is composed of m layers L_i (i = 1 ... m). Each layer is populated with integers from 0 to some specific integer $(2^p - 1)$, where p is the maximum number of bits that represent a symbol. The integers in each array are independently reordered using a sequence of random numbers r_i (i = 1, 2, ... , 2^p), where the integer at index k is swapped with the integer at the index r_k. The input to the first layer is a symbol s_i and the output is a symbol x_i indexed by the ascii value of s_i. The output of each layer L_{i-1} is first processed by diffusion action and is passed as an input to the next layer L_i.

The second processing stage is the mutation operation. This operation imposes fine-grained changes to some symbols by flipping their bits according to a mutation value defined next. The mutation operation adopts a probabilistic model for selecting which of the symbols passing through must be subjected to the mutation. In such a probabilistic model, the mutation operation is activated to handle the symbol with a probability of $\gamma \in [0, 1]$. We call γ the intensity of mutation, where $\gamma = 0$ means no symbol-mutation, while $\gamma = 1$ means all of the symbols are mutated. To effectively implement this probabilistic model; we define a list with 2^p entries. This list is populated with H $(\leq 2^p)$ replications of mutation operation, where $H = Max\ (2^{p/4}, h)$ and h is a random value. The remaining entries "$2^p - H$" are populated with the NULL operation (Idempotent operation). The content of this list is randomly scattered using a sequence of 2^prandom numbers. Given this list, the intensity of the mutation is defined by $\gamma = H/2^p$. Note, due to the random reordering of the elements in the list, the probability of selecting the mutation operation is $H/2^p$.

The third processing stage uses the noise operations to change the structure of the output sequence by reordering its symbols. The noise operations make use of two actions: Permute (or P-Action) and left shift (*L-Shift*) action. The Permute action reorders its input as described in Section 4.2. The *L-Shift* action left shifts the input sequence symbols to a number of positions.

The update handler maintains a set of M state variables that are used to perform specific actions on the re-directive lists, mutation operation, and the noise operations. Table 4 lists these state variables, their descriptions, and how they are updated. We associate a state variable V_{Li} with each layer L_i of the re-directives. These state variables are used to perform some reordering to the elements of the corresponding layer. We associate two state variables V_{M1} and V_{M2} to the mutation operation, where the first variable is used as an activator for the mutation operation and the second is used as a mutation value. We finally associate two state variables VSL and V_{Lp} to the noise operations to support its functionality.

Table 4. The state variables and their update mechanism.

Processing Stage	State Variables	Description	Update Method
Re-Directives Operation	$V_{L1}, V_{L2}, \ldots, V_{Lm}$	Each state variable V_{Li} is used to update the order of the corresponding layer L_i.	Performing an XOR operation between the state variable V_{Li} and the output of the layer L_i before the diffusion takes place.
Mutation Operation	V_{M1}, V_{M2}	V_{M1} is used to activate the mutation operation. V_{M2} is used as a mutation value	V_{M1}, V_{M2} are up-dated by XORing them with respectively the content of $L_j[l]$ and $L_{j+1}[l]$.
Noise Operations	V_{SL}, V_{LP}	V_{SL} determines the order in which the noise operations are executed. V_{LP} is the shift amount (used by the shift operation).	These two variables are refreshed by XORing their values with two random numbers.

All the state variables are initialized to 0 (zero). The update handler uses the intermediate results of the re-directives operation to continuously tune the values of the state variables (after processing each input symbol) as follows. The update handler refreshes the values of the state variables V_Li by XORing V_Li with the output of the layer L_i before the diffusion has taken place. It refreshes the values of the state variables V_{M1} and V_{M2} using the content of the layers L_i. Namely, the values of V_{M1} and V_{M2} are refreshed by XORing V_{M1} and V_{M2} with the content of $L_j[l]$ and $L_{j+1}[l]$ respectively. The indexes j and l are calculated by $j = I_k$ MOD m and $l = I_k/m$, where m is the number of layers, I_k is the symbol of the original input corresponding to the symbol that is being processed s_k, and MOD is the division remainder. The state variables V_{SL} and V_{Lp} associated with the noise operations are refreshed by XORing them with two random numbers obtained from the random generator. The rationale behind this update mechanism is to make the first two processing stages highly influenced by the input symbols, while the third processing stage masks the trace of the input symbols but maintains their impact on the output.

After defining the three processing stages of the key echo method, we describe how the key echo generation works. Suppose a key of n symbols $I_1 I_2 \ldots I_n$. The symbols are diffused using the diffusion action, yielding the new sequence $s_1 s_2 \ldots s_n$. Each symbol s_i

is subjected to successive mappings through the re-directive layers. Each layer maps its input to a new output symbol. The output symbol is used to update the corresponding state variable and then is passed to the diffusion action (using D-Action) before mapping it to the next layer. The output of the re-directives may be further manipulated by applying the mutation operation. The state variable V_{M1} is used to access the list (associated with the mutation operation). If the accessed element is NUL, no mutation is performed on the current symbol. Otherwise, the mutation operation flips bits of the input symbol by XORing this symbol with the state variable V_{M2}. Regardless of whether the mutation operation is invoked or not, the two state variables (V_{M1} and V_{M2}) must be updated as described above. The sequence of symbols is eventually passed to the noise operations. The noise operations apply the two actions: *Permutate* and *L-Shift*. The state variable V_{SL} determines the sequence in which the two operations are executed (*Permutate* $\rightarrow$ *L-Shift* or *L–Shift* $\rightarrow$ *Permutate*). Basically, the order of the execution is "*Permutate* $\rightarrow$ *L-Shift*" if V_{SL} MOD 2 = 0; the order is "*L-Shift* $\rightarrow$ *Permutate*" otherwise. The state variable V_{LP} determines the shift amount, namely we take the rightmost three bits as the amount of shift.

Before processing any new input sequences, the entries of the layers of the re-directives must be partially reordered. In particular, the layer L_i is first left shifted one position and the entry of $L_i[0]$ is swapped with the entry of $L_i[V_L i]$).

6. The Encryption Process

The encryption process uses the operations that we discussed in the previous sections to cipher blocks of plaintext $a_1a_2 \ldots a_n$. Figure 10 delineates the encryption process components and the control flow between these components. The encryption process has two fundamental subprocesses: initialization and ciphering (the numbers on the operations represent the order of the execution. Operations with the same numbers can execute in parallel). The initialization stage prepares the different inputs that are required by the ciphering operations. The initialization subprocess feeds the encryption key (n symbols) as an input to the input doubling operation, which produces $2n$ symbol sequence (Section 4). The left half of the $2n$ sequence (n symbols) becomes an input to the random number generator and the right n symbols are passed as a new input to the doubling operation to produce more $2n$ symbol sequences. The random number generator uses the seed to generate sequences of random numbers. The random shuffling operation uses these sequences of random numbers to reorder (1) the contents of the lists (i, g, Op, N_{OP}, and SWL) that are used to support the functionality of the deep masking method and (2) the contents of the lists (MUT_{OP}, L_1, L_2, ... , L_n) that support the functionality of the key round. In addition, the Swap Pattern Generation operation uses the input from the input doubling operation to generate the swapping patterns during the initialization stage.

The ciphering subprocess receives plaintext blocks $a_1a_2 \ldots a_\mathrm{n}$ as input and output-ciphered-blocks $\delta_1\delta_2 \ldots \delta_n$. The ciphering applies first the deep masking operation to the input block. This operation processes its input by first performing substitution (Section 3.1) followed by block distortion (Section 3.2). The deep masking operation iterates itself one time before it passes the intermediate output to the key round operation. The key round operation receives an input from the input doubling operation and generates the key echo $\beta_1\beta_2 \ldots \beta_n$. The key echo effect is added to the output of the deep masking operation ($\alpha_1\alpha_2 \ldots \alpha_n$) by XORing each symbol α_i with the corresponding key echo symbol β_i to yield the ultimate ciphered-block symbols δ_i.

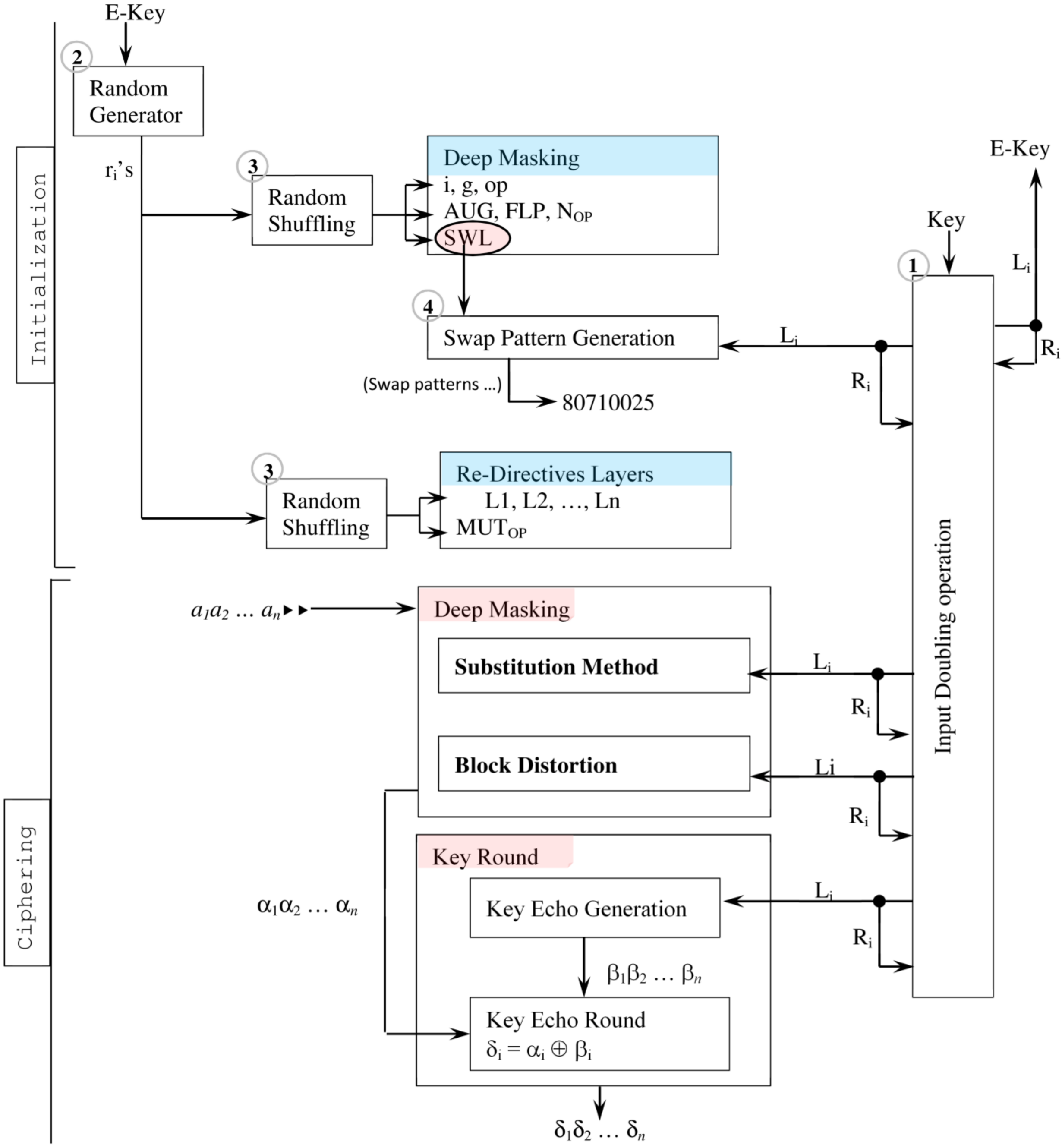

Figure 10. The encryption process.

7. The Decryption Process

The decryption process takes a ciphered block as an input and outputs the original plaintext block. Like the encryption process, the decryption process consists of two stages. The initialization stage is identical to that of the encryption process and thus we will not discuss it further. The decryption stage slightly differs in both the order of the operations execution and the operations functionality. Figure 11 shows only the part of the decryption process that needs detailed explanation.

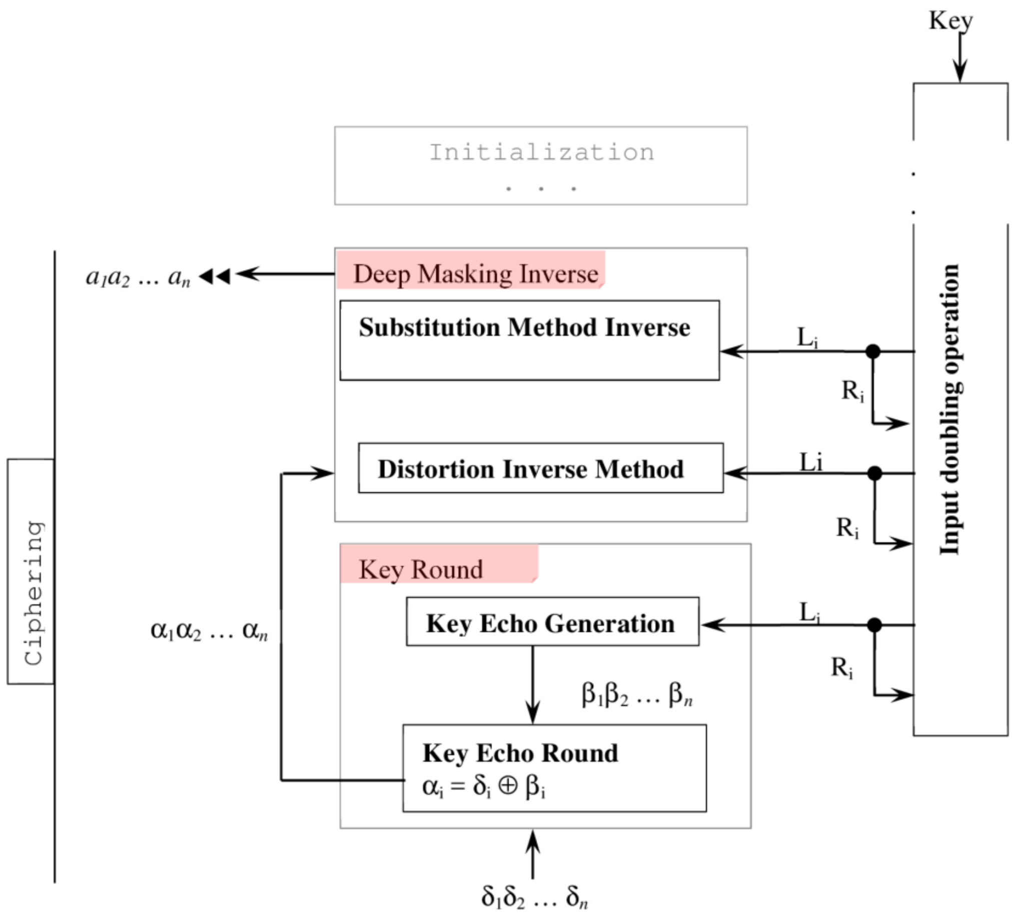

Figure 11. The Decryption process.

The decryption process executes its operations backwards: the key round first followed by the block deep distortion. The key round generates a key echo sequence $\beta_1\beta_2 \ldots \beta_n$ and XORes each β_i with the input ciphered text symbols δ_i. The outcome is the sequence $\alpha_1\alpha_2 \ldots \alpha_n$, which is passed to the inverse deep masking process for further processing. The inverse deep masking applies first the inverse distortion operations to the input and then inverse substitution operations. The outcome is the original plaintext input block $a_1a_2 \ldots a_n$.

8. Performance Analysis

The performance analysis consists of three important tasks. First, we study the security of the proposed technique by applying variety of randomness tests (Section 8.1). Second, we study the time requirement of the proposed technique (Section 8.2). Third, we discuss common cryptanalyses attacks (Section 8.3).

8.1. Security Analysis

We analyze the performance of the proposed technique in this section. In our analysis, we follow the guidelines of the NIST testing framework. To effectively test the performance, we created the following test cases as specified by NIST framework.

1. Key test case. The objective of this test is to analyze the reaction of the encryption technique to the key changes. Effective techniques should react to any change in the key by producing different and random ciphertext. To create this test case, we used 1000 different keys that only differ by one bit and one plaintext (fixing the plaintext ensures that the only changing factor is the key). All of the keys were 16 bytes, while the plaintext was 50,000 bytes.

2. Plaintext test case. This test case was used to study the reaction of the encryption technique to the plaintext changes. To create this test case, we neutralized the impact of the key by using only one key and used 1000 different plaintext sets. Similar to the key test case, all the plaintexts were 50,000 bytes and the fixed key was 16 bytes.
3. Plaintext/ciphertext correlation. The main objective of this test case is to test if there is any correlation between plaintext and its corresponding ciphertext. To create this text case, we performed an XOR operation between each plaintext (in the plaintext test case) and its corresponding ciphertext. The XOR operation was performed at the symbol level: each symbol of the plaintext was XORed with the corresponding symbol of the ciphertext.

We ran the three test cases using the same hardware and software (the encryption technique was implemented in Python 3.11). We then tested the ciphertext resulting from encrypting each test case using NIST standard randomness tests.

Tables 5–7 show the results. The results are presented in terms of the number of passed/failed sequences and the success percentage (Pass percentage).

Table 5. Key test case: NIST randomness test results.

Randomness Tests	Passed Sequences	Failed	Pass Percentage	Max Fail
Runs test	960	40	96%	105
Monobit test	960	40	96%	105
Spectral test	880	120	88%	105
Serial test	910	90	91%	105
Cumulative sums test	920	80	92%	105
Non-overlapping template matching	940	60	94%	105
Overlapping template matching	940	60	94%	105
Linear Complexity	920	80	92%	105
Approximate entropy	920	80	92%	105

Table 6. Plaintext test case: NIST randomness test results.

Randomness Tests	Passed Sequences	Failed	Pass Percentage	Max Fail
Runs test	970	30	97%	105
Monobit test	980	20	98%	105
Spectral test	900	100	90%	105
Serial test	930	70	93%	105
Cumulative sums test	890	110	89%	105
Non-overlapping template matching	960	40	96%	105
Overlapping template matching	950	50	95%	105
Linear Complexity	980	20	98%	105
Approximate entropy	960	40	96%	105

Table 7. Plaintext/ciphertext test case: NIST randomness test results.

Randomness tests	Passed Sequences	Failed	Pass Percentage	Max Fail
Runs test	980	20	98%	105
Monobit test	990	10	99%	105
Spectral test	880	120	88%	105
Serial test	910	90	91%	105
Cumulative sums test	910	90	91%	105
Non-overlapping template matching	930	70	93%	105
Overlapping template matching	920	80	92%	105
Linear Complexity	970	30	97%	105
Approximate entropy	940	60	94%	105

Referring to Tables 5 and 6, one can see that the performance of the technique is stable: the technique reacts to the changes in the plaintexts and the keys by producing random ciphertexts with high percentage. In most of the cases, we have a more than 90% pass rate. Although the spectral test (Table 5) has a pass percentage of 88%, which is lower than other tests, this percentage is still reasonably high. The results in Table 7 show that the sequences that resulted from XORing plaintext with its corresponding ciphertext are random with a high percentage. Note that the pass percentage was higher than 90% (except for Spectral test). The randomness of these XOR-created sequences indicates that the plaintext has no significant correlation with its corresponding ciphertext.

We realize that these performance numbers must be based on standard security measurements for better interpretation. Given that it is impossible to test any encryption technique for all possible inputs, NIST developed a criterion (Equation (4)) that gives assurance (with some confidence level) of whether the encryption technique is secure. The values in equation 8.1 are the number of used sequences (S), and the level of significance used (α). According to NIST recommendations, the values of α should be less than 10%.

$$Max\ Fail = S \times \left(\alpha + 3 \times \sqrt[2]{\frac{\alpha \times (1-\alpha)}{S}} \right) \tag{4}$$

Equation (4) computes an upper bound of the number of sequences (ciphertexts) that possibly fail a particular randomness test. If the number of sequences that fail each randomness test exceeds the upper limit (Max Fail), the security of the encryption technique becomes questionable. To abide by the NIST recommendation, we computed the maximum number of sequences that fail a particular test for our data sets (recall that each of our three data sets consists of 1000 sequences). We used a level of significance of 0.05. The rightmost column of each of Tables 5–7 show the results. Observe that the number of sequences that failed a particular randomness test is less than the maximum number of sequences that possibly fail as predicted by Equation (4). However, there are three incidences in Tables 5–7, where the number of failed sequences slightly exceeds the maximum. For instance, in Table 6, the number of sequences that failed the “Cumulative sums test” randomness test is 110, which slightly exceeds the maximum expected number (105).

The performance of the proposed technique is a result of the effectiveness of the constituent operations. First, although the substitution operation uses AES Sbox, the substitution operation is significantly different from the substitution technique used in AES. While the AES substitution operation is static (mapping is fully determined by the symbol itself), the substitution technique of the proposed technique is dynamic. That is, the outcome of substituting a symbol depends on the symbol itself and the state of the substitution operation, which is controlled by the move operations. Second, the deep distortion operation has a deep modification impact on its input. It utilizes several actions that perform deep bit manipulations and uses a data-dependent mechanism to specify both the applied distortion action and the manipulation pattern. Third, the key echo generation method uses a novel generation technique that produces a highly complicated key stream. This key stream significantly contributes to the security of the proposed technique.

8.2. Time Complexity Analysis

In this section, we show the time complexity of the individual operations first and then the overall complexity of the system:

1. *Key Transform*: Main steps are
   A. Forward pass: is a for loop from 1 to n, each operation in the loop is carried out in a constant time and hence the loop is $O(n \times c)$, where c is a constant. But n is very small (number of symbols in the key), hence, the loop is executed in a constant time C.
   B. Backward pass: similar to the forward pass.

   Therefore, the key transformation is carried out in a constant time.
2. *Move Operations*

All the move operation are index manipulation operations and hence are carried out in $O(1)$.

3. *Move operation selection*

The main operation here is updating the variable W (*Xor*, *Sub*, and *Shift*) then indexing the different lists Op, i and g. All of them are carried out in constant time $O(c)$.

4. *Symbol distortion methods*
   a. Mutate: is only an Xor carried out in O(1)
   b. Swap: is a loop from 1 to 8, each iteration is carried out in a constant time and hence the whole loop is carried out in a constant time.
   c. Shift operation: is a simple shift that is also carried out in a constant time.
5. *Swap pattern generation*

Is a loop from 1 to n (n usually 8). Each iteration has an *And*, *Mod* and *Shift* operation, all of which are carried out in a constant time, hence, the whole operation is carried out in a constant time.

6. *Mutate Patterns*

Mutate patters are simple random numbers, hence, they need constant time.

7. *Symbol distortion methods selection:*

The main operations here are a. updating the G^i variable b. selecting the operation from the specific lists based on Ring[G^i] value. To update G^i we use a loop from 1 to $n - 1$ (n is 8). In the loop, each iteration has an if statement and an addition or subtraction operation; both are carried ou in constant time and, hence, the whole loop is carried out in constant time. Selecting the operation is simply involves indexing the specific list and, hence, is carried out in $O(1)$.

8. *Deep masking:* It has the following steps:
   a. Select Move operation: carried out in constant time from the above.
   b. Execute Move operation: carried out in constant time from the above.
   c. Substitute operation: is an array indexing carried out in $O(1)$.
   d. Distortion operation/s selection: carried out in constant time from the above.
   e. Execute Distortion operations: carried out in constant time from the above.

All of these operations are carried out in constant time and, hence, the Deep Masking process is also carried out in a constant time.

9. *Key doubling operation* This process has the following steps:
    a. Diffusion Action: Has two loops from 1 to $n - 1$ (n number of symbols in the key). Each iteration has an *Xor* operation and a substitution, both of which require constant time and, hence, the overall operation is carried out in constant time.
    b. Mutation Action: A loop from 1 to n. Each iteration has an *Xor*, hence it requires a constant time.
    c. Augmentation Action: Similar to the Mutation Action.
    d. Permutation Action: A loop from 1 to $2n$, where the symbols are reordered. The process is carried out in constant time since n is very small (8, 16, or 32).
10. *Key echo generation*

This operation has a main loop from 1 to n (n number of symbols in key). In each iteration, we update a number of state variables. Updating the variables is carried out through arithmetic operations, Diffusions and Mutation. All the operations require constant time as shown above and, hence, the whole process is carried out in constant time.

11. *Encryption Process*

This process has one main loop from 1 to n (n is the number of bytes in a block of data). In each iteration, a data symbol (character), is deeply masked, a key echo is generated, and the deeply masked output is Xored with the key echo. Deep masking requires constant time as shown above. Key echo also requires a constant time. The Xor operation is also carried out in a constant time. Hence, all three actions in each iteration require a constant time c, then the encryption process is carried out in $O(c \times n)$ and, hence, in linear time.

8.3. Common Cryptanalysis Attacks

Strong ciphers are designed in a way that makes it extremely hard for a cryptanalyst to break them. In this section, we discuss how the proposed design is secure against common cryptanalysis attacks.

8.3.1. Known-Plaintext Analysis (KPA)

By knowing parts of the plaintext and their ciphertext and by using reverse engineering, the attacker tries to recover the key and use it to decipher the rest of the ciphertext. This attack may work when one key is used to cipher the whole text. In the proposed cipher, each individual plaintext symbol is encrypted with a separate key (Key Echo) and, hence, it is immune to this attack.

8.3.2. Chosen-Plaintext Analysis (CPA)

By choosing random plaintexts and obtaining the corresponding ciphertext, the attacker tries to recover the key. Similar to KPA, knowing one key will not allow the attacker to decrypt the rest of the ciphertext as other key echoes are used in the encryption process.

8.3.3. Ciphertext-Only Analysis (COA)

The attacker knows only the ciphertext and needs to recover both the key and the plaintext. This attack is very hard but most probable. Figures 4–6 show that our cipher produces excellent randomness results, which indicates that the relation between plaintext and ciphertext symbols is very random. The results indicate that recovering a plaintext symbol from a ciphertext symbol is extremely hard.

8.3.4. Brute-force Attacks

Trying out all possible key values is referred to as brute-force attack. This attack works well on short keys but is unfeasible for longer keys (using current computing infrastructure). In the proposed algorithm, we do not impose a limit on the length of the key. It could be 256, 512 or more bits and hence it is safe here as well.

8.3.5. Differential Cryptanalysis

This attack is a type of CPA attack, where the attackers monitor a number of plaintext parts and analyze how they transform into ciphertext, hoping to deduce the key. Since we use multiple key echoes to encrypt, our cipher can effectively withstand these type of attacks.

8.3.6. Linear Cryptanalysis

In this attack a number of KPAs is performed on a number of messages that were encrypted using the same key. The more messages the attacker has the higher the probability of finding a key. In our approach, we never use the original key to encrypt with. We use key echoes instead. Knowing a key echo will not reveal the original key due to the fact that we use chaotic random numbers to produce key echoes, and we use the Transform function (Figure 1) to seed the chaotic random number generators. The Transform function produces a random value from the original key. To obtain the original key from a key echo, the attacker needs to know the sequence of random numbers leading back to the seed and then from the seed, the attacker needs the inverse of the Transform function to obtain the original key.

8.3.7. Side Channel Attacks

The attackers here monitor power consumption, radiation emission and/or time of data processing. This technique may reveal some information leading to the key if the cipher is not well designed. In the proposed system, although we use plaintext dependent actions, we made sure that processing is unified. That is, regardless of the symbol we are processing, time and power consumption are almost identical. All relevant loops are iterated on all bits of the symbol (8 for example) and all the branches in any loop require a similar amount of time as they are almost identical. In the dynamic substitution processes, deciding the substituent requires the same amount of time and power regardless of the input symbol. In addition to that, accessing the substituent cell is the same regardless of the input. Additionally, selecting the noise patterns and adding noise to an input symbol require the same amount of time and power regardless of the input symbol. We strongly believe that this type of attack will not reveal any useful information to the attacker. However, what if it did? The attacker may obtain information related to some key echoes only. Recovering the original key is extremely hard, as we have shown above (Section 8.3.6).

9. Conclusions

We proposed in this paper an encryption technique. The technique has many powerful processing operations that effectively transform its input (plaintext) to a random uncorrelated ciphertext. In particular, the technique has a substitution technique whose functionality depends not only on the input symbols but also on the set of distortion operations that make the substitution nondeterministic. The technique is provided with masking operations that increase the confusion and the avalanche effect. Importantly, the functionality of the masking operations and their selection is carried out using a data dependent mechanism. This means that any change in the input imposes changes to the resulting ciphertext. The key echo generation process uses effective techniques and an expansion method that produce powerful key codes for further hiding the ciphertext and hiding the key identity. These features make the proposed technique highly dynamic and very sensitive to the changes in the input.

The security tests showed that the technique is powerful and secure. As Tables 5–7 show, the output of the technique is random as a high percentage of the sequences passed the standard NIST randomness tests. The technique is also time efficient. Our complexity analysis indicates that the technique has linear complexity with the size of the input, making it suitable for systems that need high speed.

The proposed algorithm can be utilized in a number of applications. For example, it is very suitable for any application that runs on a smart device such as smartphone or an

IOT device with limited hardware. The current version is implemented in Python, which makes time comparison with other Ciphers unfair.

Due to time constraints, we left a few issues for future work. First, we plan to implement an optimized version of the proposed cipher using an efficient language such as *C* language and compare its performance with state-of-the-art algorithms such as AES. Second, we intend to review all constants and functions to make sure that they use "nothing up my sleeve numbers". Third, we also hope to review the applicability of the impossible differential attack (used against ciphers with multiple rounds; ours use only one round) and the algebraic attack (used mainly against stream ciphers).

Author Contributions: Conceptualization, M.J.A.-M.; methodology, M.J.A.-M.; software, A.A.A.-D.; validation M.J.A.-M. and A.A.A.-D.; formal analysis, M.J.A.-M.; investigation A.A.A.-D.; writing—original draft preparation, M.J.A.-M.; writing—review and editing, M.J.A.-M. and A.A.A.-D.; visualization, M.J.A.-M.; supervision, M.J.A.-M. and A.A.A.-D.; project administration, M.J.A.-M. and A.A.A.-D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the American University of Madaba, Jordan.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data are available in the document.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Al-Muhammed, M.J.; Abuzitar, R. Intelligent Convolutional Mesh-Based Encryption Technique Augmented with Fuzzy Masking Operations. *Int. J. Innov. Comput. Inf. Control.* **2020**, *16*, 257–282.
2. Al-Muhammed, M.J.; Abuzitar, R. κ–Lookback Random–based Text Encryption Technique. *J. King Saud Univ. Comput. Inf. Sci.* **2019**, *31*, 92–104. [CrossRef]
3. Hendricks, J.; Burke, B.; Gamage, T. Polysizemic Encryption: Towards a Variable-Length Output Symmetric–Key Cryptosystem. In Proceedings of the IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Milwaukee, WI, USA, 15–19 July 2019; Volume 2, pp. 688–693.
4. Daemen, J.; Rijmen, V. The Advanced Encryption Standard Process. In *The Design of Rijndael, Information Security and Cryptography (Texts and Monographs)*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 1–8.
5. Azam, N.A. A Novel Fuzzy Encryption Technique Based on Multiple Right Translated AES Gray S–Boxes and Phase Embedding. *Secur. Commun. Netw.* **2017**, *2017*, 9. [CrossRef]
6. Modi, B.; Gupta, V. A Novel Security Mechanism in Symmetric Cryptography using MRGA. In *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*; Sa, P., Sahoo, M., Murugappan, M., Wu, Y., Majhi, B., Eds.; Springer: Singapore, 2018; Volume 719, pp. 195–202.
7. Biham, E.; Anderson, R.; Knudsen, L. Serpent: A Proposal for the Advanced Encryption. Available online: https://www.cl.cam.ac.uk/rja14/serpent.html (accessed on 1 January 2020).
8. Patil, P.; Narayankar, P.; Narayan, D.G.; Meena, S.M. A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish. *Procedia Comput. Sci.* **2016**, *78*, 617–624. [CrossRef]
9. Han, S.J. The Improved Data Encryption Standard (DES) Algorithm. In Proceedings of the IEEE 4th International Symposium on Spread Spectrum Techniques and Applications, Mainz, Germany, 25 September 1996; pp. 1310–1314.
10. Wang, L.; Zhang, Y. A New Personal Information Protection Approach based on RSA Cryptography. In Proceedings of the 2011 IEEE International Symposium on IT in Medicine and Education, Cuangzhou, China, 9–11 December 2011; Volume 1, pp. 591–593.
11. Rivest, L.R.; Robshaw, M.J.B.; Sidney, R.; Yin, Y.L. The RC6 Block Cipher. In Proceedings of the First Advanced Encryption Standard (AES) Conference, Ventura, CA, USA, 20–22 August 1998.
12. Weiping, P.; Danhua, C.; Cheng, S. One-Time-Pad Cryptography Scheme based on a Three–Dimensional DNA Self-Assembly Pyramid Structure. *PLoS ONE* **2018**, *13*, e0206612.
13. Cui, G.; Han, D.; Wang, Y.; Wang, Z. An Improved Method of DNA Information Encryption. In *Bio-Inspired Computing-Theories and Applications*; Pan, L., P´aun, G., Pèrez-Jiménez, M.J., Song, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 472, pp. 73–77.
14. ElKamchouchi, D.H.; Mohamed, H.G.; Moussa, K.H. A Bijective Image Encryption System Based on Hybrid Chaotic Map Diffusion and DNA Confusion. *Entropy* **2020**, *22*, 180. [CrossRef] [PubMed]

15. Babaei, M. A Novel Text and Image Encryption Method based on Chaos Theory and DNA Computing. *Nat. Comput.* **2013**, *12*, 101–107. [CrossRef]
16. Zhang, Y.; Wang, Z.; Wang, Z.; Liu, X.; Yuan, X. A DNA-Based Encryption Method Based on Two Biological Axioms of DNA Chip and Polymerase Chain Reaction (PCR) Amplification Techniques. *Chem. A Eur. J.* **2017**, *23*, 13387–13403. [CrossRef] [PubMed]
17. Thangavel, M.; Varalakshmi, P.; Sindhuja, R.; Sridhar, S. Towards Secure DNA Based Cryptosystem. In *Data Science Analytics and Applications*; Sharma, R.S., Ed.; Springer: Singapore, 2018; Volume 804, pp. 163–177.
18. UbaidurRahmana, N.H.; Balamuruganb, C.; Mariappanc, R. A Novel DNA Computing based Encryption and Decryption Algorithm. *Procedia Comput. Sci.* **2015**, *2015*, 463–475. [CrossRef]
19. Halvorsen, K.; Wong, W. Binary DNA Nanostructures for Data Encryption. *PLoS ONE* **2012**, *7*, e44212. [CrossRef] [PubMed]
20. Kalsi, S.; Kaur, H.; Chang, V. DNA Cryptography and Deep Learning using Genetic Algorithm with NW algorithm for Key Generation. *J. Med. Syst.* **2018**, *42*, 1–17. [CrossRef] [PubMed]
21. Clelland, C.T.; Risca, V.; Bancroft, C. Hiding Messages in DNA Microdots. *Nature* **1999**, *399*, 533–534. [CrossRef] [PubMed]
22. Juels, A.; Ristenpart, T. Honey encryption: Security Beyond the Brute-Force Bound. In *Advances in Cryptology–EUROCRYPT*; Nguyen Phong, Q., Elisabeth, O., Eds.; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8441, pp. 293–310.
23. Oludare, O.; Aman, J.; Oludare, A. A Comprehensive Review of Honey Encryption Scheme. *TELKOMNIK-A Indones. J. Electr. Eng.* **2019**, *13*, 649–656.
24. Yin, W.; Indulska, J.; Zhou, H. Protecting Private Data by Honey Encryption. *Secur. Commun. Netw.* **2017**, *2017*, 9. [CrossRef]
25. Yoon, J.W.; Kim, H.; Jo, H.J.; Lee, H.; Lee, K. Visual Honey Encryption: Application to Steganography. In Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security, Portland, OR, USA, 17 June 2015; pp. 65–74.
26. Iavich, M.; Gnatyuk, S.; Jintcharadze, E.; Polishchuk, Y.; Odarchenko, R. Hybrid Encryption Model of AES and ElGamal Cryptosystems for Flight Control Systems. In Proceedings of the 5th International Conference on Methods and Systems of Navigation and Motion Control (MSNMC), Kiev, Ukraine, 16–18 October 2018; pp. 229–233.
27. Li, X.; Yu, L.; Wei, L. The application of hybrid encryption algorithm in software security. In Proceedings of the 3rd International Conference on Consumer Electronics, Communications and Networks, Xianning, China, 20–22 November 2013; pp. 669–672.
28. Goyal, V.; Kant, C. An Effective Hybrid Encryption Algorithm for Ensuring Cloud Data Security. In *Big Data Analytics*; Aggarwal, V., Bhatnagar, V., Mishra, D., Eds.; Springer: Singapore, 2018; Volume 654, pp. 195–210.
29. Ren, W.; Miao, Z. A Hybrid Encryption Algorithm Based on DES and RSA in Bluetooth Communication. In Proceedings of the 2nd International Conference on Modeling, Simulation and Visualization Methods, Sanya, China, 15–16 May 2010; pp. 221–225.
30. Cheng, H.; Zheng, Z.; Li, W.; Wang, P. Probability Model Transforming Encoders Against Encoding Attacks. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 28 May 2019.
31. Courtois, N.T.; Pieprzyk, J. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In *Advances in Cryptology ASIACRYPT 2002. ASIACRYPT 2002*; Zheng, Y., Ed.; Springer: Berlin, Heidelberg, Germany, 2002; Volume 2501, pp. 267–287.
32. Dewu, X.; Wei, C. A Survey on Cryptanalysis of Block Ciphers. In Proceedings of the 2010 International Conference on Computer Application and System Modeling (ICCASM 2010), Taiyuan, China, 22–24 October 2010; Volume 8, pp. 218–220.
33. Tiessen, T. Secure Block Ciphers–Cryptanalysis and Design. Ph.D. Thesis, Technical University of Denmark, Lyngby, Denmark, 2017.
34. Liu, Y. Techniques for Block Cipher Cryptanalysis. Ph.D. Thesis, Katholieke Universiteit Leuven, Belgium, 2018.
35. Dou, Y.; Liu, X.; Fana, H.; Liad, M. Cryptanalysis of a DNA and Chaos Based Image Encryption Algorithm. *Optik* **2017**, *145*, 456–464. [CrossRef]
36. Chai, X.-L.; Gan, Z.-H.; Yuan, K.; Lu, Y.; Chen, Y.-R. An Image Encryption Scheme based on Three–Dimensional Brownian Motion and Chaotic System. *Chin. Phys. B* **2017**, *26*, 020504. [CrossRef]
37. Khan, M.; Masood, F.; Alghafis, A.; Amin, M.; Naqvi, S.I.B. Batool. A Novel Image Encryption Technique using Hybrid Method of Discrete Dynamical Chaotic Maps and Brownian Motion. *PLoS ONE* **2019**, *14*, e0225031. [CrossRef] [PubMed]
38. Gan, Z.; Chai, X.; Zhang, M.; Lu, Y. A Double Color Image Encryption Scheme based on Three-Dimensional Brownian Motio. *Multimed. Tools Appl.* **2018**, *77*, 27919–27953. [CrossRef]
39. Stallings, W. *Cryptography and Network Security: Principles and Practice*, 8th ed.; Pearson: London, UK, 2019.