Light but Effective Encryption Technique based on Dynamic Substitution and Effective Masking

Muhammed Jassem Al-Muhammed
Faculty of Information Technology
American University of Madaba
Madaba, Jordan

***Abstract*—Cryptography and cryptanalysis are in ever-lasting struggle. As the encryption techniques advance, the cryptanalysis techniques advance as well. To properly face the great danger of the cryptanalysis techniques, we should diligently look for more effective encryption techniques. These techniques must properly handle any weaknesses that may be exploited by hacking tools. We address this problem by proposing an innovative encryption technique. Our technique has unique features that make it different from the other standard encryption methods. Our method advocates the use of dynamic substitution and tricky manipulation operations that introduce tremendous confusion and diffusion to ciphertext. All this is augmented with an effective key expansion that not only allows for implicit embedment of the key in all of the encryption steps but also produces very different versions of this key. Experiments with our proof-of-concept prototype showed that our method is effective and passes very important security tests.**

***Keywords*—Encryption techniques; dynamic substitution; key expansion; directive based manipulation; block masking**

I. Introduction

In the digital era, almost all of our sensitive information is either transmitted over the network or digitally stored on machines. This information will inevitably be in a great risk if we do not properly secure them. Encryption is the *de-facto* means for keeping the security of the transmitted or stored information. Many encryption techniques have been proposed (e.g. [1][2][3][4][5][6][7][8][9][10] [11][12][13][14][15][16][17][18]). Although these methods are effective and purport to provide high levels of security, there is always a truly pressing need for new techniques that can effectively face the formidably ever-advancing hacking tools.

This paper proposes an effective encryption technique. This technique consists of three effective encryption operations along with a novel key expansion operation. First, the proposed substitution operation adopts dynamic behavior. Unlike all other encryption techniques (noticeably [3]), which use a static substitution operation, our technique uses a dynamic substitution operation whose state depends on the key and greatly sensitive to its changes. Second, the diffuse operation is highly sensitive to changes of the key or the plaintext block and is capable of greatly magnifying and then propagating these changes to all of the block's symbols. Third, the masking operation uses a novel technique that makes sharp changes to both the individual symbols of the plaintext block and to its structure. Finally, the key expansion technique proposes an innovative model that combines the lookback-based substitution with the random manipulations.

The paper makes the following contributions. First, it proposes a full-fledged encryption technique. Second, it proposes an effective dynamic substitution operation whose functionally highly depends on the key. Third, it proposes a novel key expansion technique. Forth, it proposes an effective masking operation that impacts both the block's individual symbols and the structure of the block.

We present our contribution as follows. Section II describes the technical details of the dynamic substitution operation. Sections III through VI present the details of the operations that comprise our proposed cipher. Section VII presents the technical details of the proposed cipher. Section VIII presents the related work. Section IX presents the performance analysis of our technique. We conclude and give directions for future work in Section X.

II. Dynamic Box Substitution

The purpose of the substitution is to move from the actual block symbols to new symbols. It uses a data structure superficially similar to S–Box of AES encryption method [3], but is fundamentally different in its dependency on the key. This section describes the D–Box and discusses how it is used in substituting symbols.

A. The D-Box and its Inverse D-Box^{-1}

The D-Box is conceptually $K \times K$ array. The D-Box is populated with the unicode symbols from 0 to K^2-1. Each symbol in the D-Box can be accessed by its column and row indexes. The positions of the unicode symbols in the D-Box are never static. Their locations depend on the encryption key. To do this, we utilize a sequence of integers $I_1 I_2 ... I_m$, which are generated in a process that involves the key. (The process for generating these integers is discussed in Appendix A.) These integers are used to reposition the symbols in the D-Box. The repositioning of the symbols is performed by swapping the symbol at the index i in the D-Box with the symbol at the index I_i.

In order to use the D-Box for encryption, it is necessary to define the inverse substitution so that the original block can be recovered. Fig. 1 illustrates the process of creating D-Box^{-1} from the D-Box. Given a 16×16 D-Box, the D-Box^{-1} is created as follows. For each symbol s in D-Box[r, c], we create an entry in the D-Box^{-1} by dividing the 8 bits of s

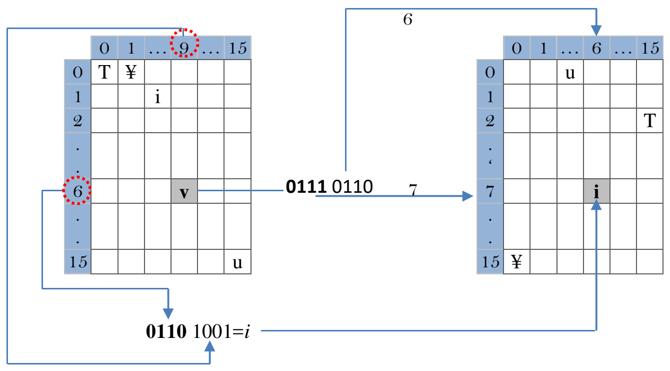


Fig. 1. the creation of D-Box^{-1} (right) from D-Box (left).

into two halves.[1] The left half bits create an index x for the row of D-Box^{-1} and the right half bits create an index y for the column of D-Box^{-1}. On the other hand, the 4 bits that represent r and the 4 bits that represent c are concatenated rc to create a unicode symbols t. The unicode symbol t is placed in the index (x, y) of D-Box^{-1}.

Referring to Fig. 1, the symbol **v** is at the index r = 6, c = 9. The 8 bits that represent **v** are "01110110" (118 in decimal). The left half of the bits "**0111**" (7 in decimal) index the row of the D-Box^{-1} and the right 4 bits "**0110**" (6 in decimal) index its column. On the other hand, the row index of **v** in D-Box is r = 6 ("**0110**") and the column index of **v** in D-Box is c =9 ("**1001**"). The binaries of the row index and the column index are concatenated "**01101001**" to create the entry **i** (105 in decimal). The symbol **i** is placed in the D-Box^{-1} [7, 6] as shown in Fig. 1.

B. Symbol-Substitution using D-Box / D-Box^{-1}

We define two substitution operations. The first operation is called **Substitute**, which uses the D-Box for replacing a symbol p with a new one. The second operation is called **Inverse Substitute**, which replaces a symbol q using D-Box^{-1}. Substituting a symbol using the D-Box and D-Box^{-1} is straightforward. It is a table lookup operation. Given a symbol p, the left half bits of p's binary representation index the row of the D-Box and the right half bits index its column. Indexing D-Box^{-1} is done in an identical way.

Fig. 2 shows an example of D-Box and its Inverse D-Box^{-1} (right). For the sake of the simplicity, we use only the hexadecimal numbers. Since we have only 16 symbols, our D-Box is 4×4. Therefore, we need two bits to index its rows and two bits to index its columns. The two bits at the top index the column and the two bits on the leftmost index the rows.

To illustrate the substitution operation, consider a block of four symbols "9C06". The **Substitute** operation reads the first symbol "9" whose binary representation is "1001". The left two bits "10" index the row of the D-Box and the right two bits "01" index the column. The symbol at row 2 ("10") and column 1 ("01") in the D-Box is retrieved as a substitute for the symbol "9". Thus, the symbol "9" is substituted with "7". Continuing likewise, the entire block is substituted yielding the new block "7F4B".

[1]We assume, without losing the generality, K=16. The D-Box is therefore 16×16. Since we use 16×16 D-Box, the rows are indexed by 4 bits and the columns by 4 bits. Additionally each symbol in the D-Box is represented by 8 bits.

D-Box

	00	01	10	11
00	**4**	**A**	**6**	**3**
01	**2**	**9**	**B**	**C**
10	**5**	**7**	**E**	**8**
11	**F**	**0**	**1**	**D**

D-Box^{-1}

	00	01	10	11
00	**D**	**E**	**4**	**3**
01	**0**	**8**	**2**	**9**
10	**B**	**5**	**1**	**6**
11	**7**	**F**	**A**	**C**

Fig. 2. D-Box and its inverse D-Box^{-1}.

```
Input Key: $k_1k_2...k_n$
Output: manipulated key $\mathcal{W}$
$\mathcal{T}_1$= Substitute ($k_1$)
$\mathcal{W}$ = $\mathcal{W}$ + $\mathcal{T}_1$
FOR i = 2 to n DO
    $K_i^\star$ = $\mathcal{T}_{i-1} \oplus K_i$
    $\mathcal{T}_i$ = Substitute($K_i^\star$)
    $\mathcal{W}$ = $\mathcal{W}$ + $\mathcal{T}_i$
RETURN $\mathcal{W}$
```

Fig. 3. the substitution operation for a key's symbols.

To recover the original block from "7F4B", we use the **Inverse Substitute** operation. The binary of the symbol "7" is "0111". The left half bits "01" index the row of the D-Box^{-1} and the right half bits "11" index its column. As a result, the symbol at row 1 and column 3, which is "9", is retrieved from the D-Box^{-1}. The next symbol is "F" whose binary representation is "1111". Thus, the symbol at row "11" and column "11", which is "C", is retrieved from the D-Box^{-1}. Using the same process, we recover the original block "9C06".

III. Key Expansion

The key expansion process expands encryption keys to an arbitrary length. The process defines two operations. The first one is the *substitution* operation whose logic is fully described by Fig. 3. The input to this operation is a key, which consists of n unicode symbols $k_1k_2...k_n$. The output is a new (substituted) key $\mathcal{W}$ whose length is also n symbols. As the logic clearly shows, the first symbol k_1 in the key is substituted with a new one using the D-Box. For all the symbols k_i (i=2, 3, ..., n), the operation uses the outcome of substituting k_{i-1} (or $\mathcal{T}_{i-1}$) to substitute the symbol k_i. Therefore, to create a new symbol in the output key $\mathcal{W}$, the operation XORes the current symbol of the input key k_i and the result of the previous substitution $\mathcal{T}_{i-1}$ to create a new symbol $K_i^\star$. The symbol $K_i^\star$ is substituted to produce the new symbol $\mathcal{T}_i$ in the output key.

The second operation, called *Manipulate*, creates sharp modifications to its input string. The modifications include changes to both individual symbols and the structure of the input string. Suppose $t_1t_2...t_n$ be an input string. The manip-

$$Manipulate(t_i) = \begin{cases} \overset{LH}{Flip}(t_i) & if\ t_i \le t_{i+1} \\ \overset{RH}{Flip\,Swap}(t_i) & if\ t_i > t_{i+1} \end{cases}$$

Fig. 4. the manipulation operation.

```
Manipulate (tᵢ)
  1. Generate a random $\gamma_i \in (0, 1)$
  2. If ($\gamma_i \le$ LTE) THEN execute Flip action.
  3. Else execute FlipSwap action.
```

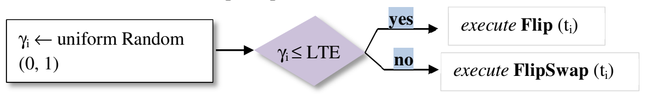


Fig. 5. the dynamic random model for action selection.

ulate operation handles each symbol t_i using the two actions defined in Fig. 4.

The manipulate operation (Fig. 4) handles each symbol t_i based on the lookahead symbol t_{i+1}. If symbol t_i lexically comes before or equal to the lookahead symbol t_{i+1}, the left half bits of t_i are flipped using the action $\overset{LH}{Flip}(t_i)$. If otherwise, the right half bits of t_i are flipped and the resulting symbol is moved to a position determined by the lookahead symbol t_{i+1}. Consider for instance the input string "pqetreeloopc". If the current symbol is "p", then the left half bits of "p" (**0111**0000) are flipped because "p" lexically comes before the lookahead symbol "q" yielding "€" (10000000). Assume that the current symbol is "q". The lookahead symbol is thus "e". Since "q" lexically comes after "e", the right half bits of "q" (0111**0001**) are flipped (0111**1110**) yielding "∼". Additionally and based on Fig. 4, the resulting symbol "∼" is moved to the position 101 % 12 = 5 in the string. (101 is the unicode index of the lookahead symbol "e" and 12 is the length of the input string.) This operation results in "€...∼.......".

Although selecting and applying one of the two actions of the *Manipulate* operation based on the lexical order of input symbols may be effective by itself, we prefer to add more randomness to the action selection. As such, instead of using the plain lexical order of the input symbols, we introduce a random noise to the action selection. In particular, we define a dynamic random model that guides the selection process (Fig. 5). Let *T* be the number of the so-far processed symbols and *P* be the number of times in which $t_i \le t_{i+1}$. Based on this, we define two dynamically updated variables *LTE* and *GT* as follows.

$$LTE = \frac{P}{T} \quad and \quad GT = 1 - \frac{P}{T}$$

Where *LTE* means "less than or equal" and represents the ratio in which the condition $t_i \le t_{i+1}$ holds. *GT* means "greater than" and represents the ratio in which the condition $t_i \le t_{i+1}$ is false. Using these two variables, we redefine our *Manipulate* operation (Fig. 4) in Fig. 5.

Although neither *LTE* nor *GT* is random, their values change (increase or decrease) based on the lexical order of the current symbols b_i and the lookahead symbols b_{i+1}. The amount of the bias toward either action therefore does change. When *LTE* increases (*GT* decreases), the likelihood of executing the *Flip* action becomes larger than the likelihood of executing *FlipSwap* action. Likewise, when *GT* increases (*LTE* decreases), the likelihood of executing the *FlipSwap* action becomes larger than that of executing *Flip* action. As such, *LTE* increases (or decreases) the likelihood of the executing an action over the other, but the choice of the action to be executed happens randomly since it depends on the random value γ_i. For instance, if *LTE* is 0.8, the likelihood of executing *Flip* is much higher than that of executing *FlipSwap*, but the actual choice whether to execute *Flip* or *FlipSwap* depends on the current random value γ_i.

```
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
```

Fig. 6. the key expansion process.

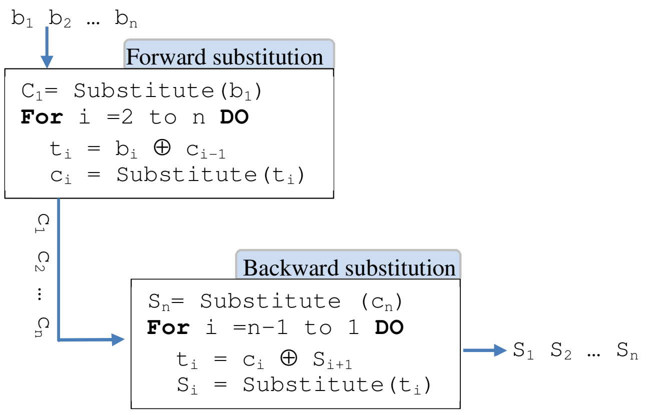
After introducing the two operations that constitute the key expansion process, we delineate this process in Fig. 6. The logic is straightforward. We start with the input key $k_1k_2...k_n$. The process creates a new version of the input key using the steps 1 through 7. The symbols of the input key are first substituted with new symbols x_i's. Next, the new symbols x_i's are manipulated using the manipulation operation (Fig. 5). To define the amount of the bias *LTE* (line 6), we always increment the number of processed symbols *T* while incrementing *P* only when the currently processed symbols x_i lexically equals to or comes before the lookahead symbols x_{i+1}. Therefore, the ratio *LTE* is always in the interval [0, 1] and represents a likelihood of executing *Flip* action.

The new version of the key *S* is concatenated with the original key. If the desired length has not been reached yet (step 9), the process uses the latest version *S* of the key to create a new version.

IV. Diffuse and Inverse Diffuse Operations

The **diffuse** operation detects changes in the input block and propagates this change to affect every symbol in the corresponding output block. To be effective, the diffuse operation must be highly sensitive to the input's change regardless of its magnitude and amplify it so that this change causes tremendous changes to the output. The **inverse diffuse** operation reverses the effect of the diffuse operation and recovers the original input.

Fig. 7 shows the algorithmic steps for the diffuse operation. As the figure shows, it performs double substitutions for the



```
C1= Substitute(b1)
For i =2 to n DO
    t_i = b_i ⊕ c_{i-1}
    c_i = Substitute(t_i)
```

```
Sn= Substitute (cn)
For i =n-1 to 1 DO
    t_i = c_i ⊕ S_{i+1}
    S_i = Substitute(t_i)
```

Fig. 7. the algorithmic steps of the diffuse operation.

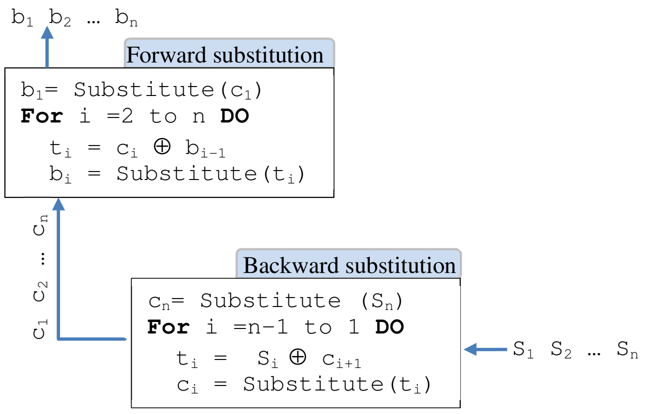
block's symbols: forward and backward substitutions. When operates on an input block $b_1b_2...b_n$, the forward substitution effectively propagates the change in a symbol b_i to all the following symbols b_j ($j > i$). First, the operation substitutes b_1 to yield a new symbol c_1. For every subsequent input symbol b_i ($i > 1$), the operation first XORes b_i with the result of the previous substitution c_{i-1} and substitutes the outcome of the XOR. That is, the symbol t_i is calculated as $t_i = b_i \bigoplus c_{i-1}$ and then substituted to yield a new symbol c_i.

As Fig. 7 shows, the output of the forward substitution $c_1c_2...c_n$ is passed to the backward substitution. The backward substitution uses similar logic as the forward except that it starts from the end of the input block. The backward substitution substitutes c_n to yield the output symbol S_n. For the input symbols c_i (i=n-1, n-2, ..., 1), c_i is first XORed with S_{i+1} and the result of the XOR operation is substituted to yield S_i.

With this feedback-based forward substitution, the result of substituting a symbol b_i is impacted not only by the symbol b_i per se, but also by the symbol b_{i-1}. That is because if a symbol b_i changes so does its substitution outcome. This change also impacts the substitution of the following symbol b_{i+1} due to the XORing operation, which in turn impacts the substitution of b_{i+2}, and so on. In other words, the change in symbol b_i collectively creeps to affect all the successive symbols b_j ($j > i$). Similarly, the backward substitution propagates the change in the symbol b_i back to the symbols b_k ($k < i$). In this case, no matter where the change occurs, the forward and backward substitutions always guarantee that this change impacts every symbol of the input.

The diffuse operation can be reversed. Fig. 8 shows the algorithmic steps of the inverse diffuse operation. As the figure shows, the operation starts from the backward substitution to recover the original block $b_1b_2...b_n$. The input to backward substitution is $S_1S_2...S_n$. The backward operation yields the block $c_1c_2...c_n$. This block is passed as an input to the forward substitution operation, which recovers the original block.

Table I shows an example of the diffuse operations output. The input blocks in the first three rows differ in only a single bit. This minor change causes a very remarkable difference in the corresponding output blocks. Additionally, the position of



```
b1= Substitute(c1)
For i =2 to n DO
    t_i = c_i ⊕ b_{i-1}
    b_i = Substitute(t_i)
```

```
cn= Substitute (Sn)
For i =n-1 to 1 DO
    t_i = S_i ⊕ c_{i+1}
    c_i = Substitute(t_i)
```

Fig. 8. the algorithmic steps of the inverse diffuse operation.

TABLE I. A SAMPLE OF THE DIFFUSE OPERATION'S OUTPUT

Input Block	Output Block
aaaaaaaaaaaaaaaa	A8 **9B** 32 **81** D1 **91** B9 **EE** E5 **60** 1F **A6** 50 **0E** 2B **36**
aaaaaaaaaaaaaaa**b**	59 **2D** B9 **BE** AB **44** 0A **88** 35 **7A** 01 **B4** D7 **8A** 08 **CD**
aa**b**aaaaaaaaaaaaa	F3 **3D** 00 **42** 80 **EF** D9 **0F** D8 **93** E3 **55** DC **FA** 0A **2A**
To be or not to be	2A **0F** 5B **51** EA **E9** 0B **41** **DC** **D1** 1C **30** 58 **1C** 31 **9B** 10 **76**
To **B**e or not to be	B5 **71** C4 **1F** A3 **4F** D9 **27** D9 **45** E0 **36** 86 **FE** 6C **08** E8 **19**

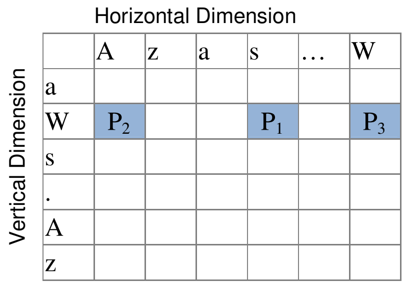


Fig. 9. the mesh.

the change does play a major role. As an example, compare between the output in row 1 and the second and third rows.

V. DIRECTIVE GENERATOR

This section describes the directives generator. Subsection V-A discusses the mesh and Subsection V-B shows how to use the mesh to produce directives.

A. The Mesh

Conceptually the mesh is a two dimensional array. The horizontal and vertical dimensions are populated with the unicode symbols from 0 to some integer N. Fig. 9 provides an example of a mesh.

Each cell in the mesh is accessed by its row and column indexes. Each move within the mesh whether along the vertical or horizontal dimension has a distance and a direction with respect to the current position. The distance of the move d is the number of cells passed. The direction of the move is either toward a lower or a higher index. We capture the move toward the lower indexes by the flag "–" and toward the higher indexes by the flag "+" regardless whether this move is along the horizontal or vertical dimension. For example, if we move from the current point, say P_1, to P_2, the distance of the move

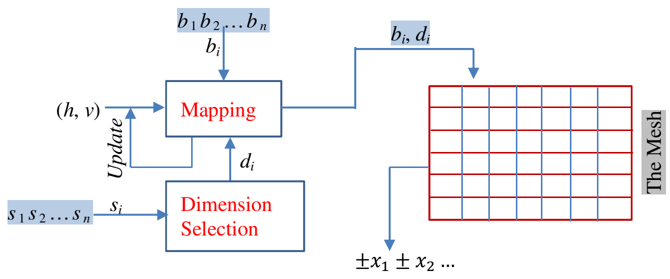


Fig. 10. mapping using the mesh.

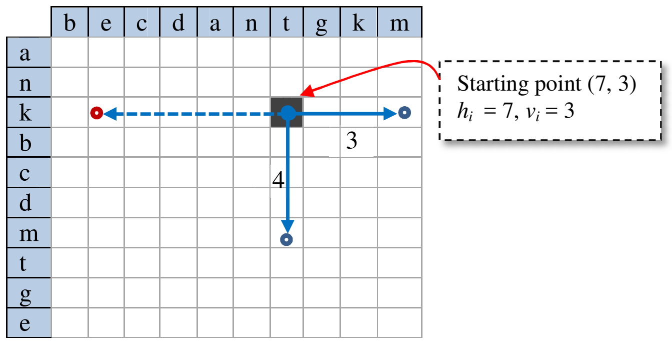


Fig. 11. example of mapping the symbol *m*.

is 3 and the direction is captured by "–" because we moved three cells to the lower indexes. As another example, the move from the current point P_1 to P_3 has a distance of 2 and the direction is "+" because we moved two cells to the higher indexes.

We can now formally define a directive as a signed integer $\pm\ x$, where $x \geq 0$ represents the distance of the move and $\pm$ represents its direction. Accordingly, moving from the current point P_1 to P_2 is captured by the directive "–3" and from P_1 to P_3 is captured by the directive "+2".

B. Mesh-Based Mapping

Based on the mesh and directive definition, the proposed mesh-based mapping consists of two operations: Mapping and Dimension Selection (see Fig. 10). The mapping operation utilizes three inputs: a block of symbols to be mapped $b_1b_2...b_n$, a mapping dimension d_i (could be the horizontal or the vertical dimension of the mesh), and a starting point (*h*, *v*). The starting point (*h*, *v*) is a point within the boundary of the mesh and from which the mapping is started; where *h* represents the index on the horizontal dimension and *v* represents the index on the vertical dimension.

Initially, the starting point is created from the first symbol of the input b_i's and the first symbol of the input s_i's. That is, the starting point is defined as (s_1, b_1). The mapping operation updates the starting point as the mapping proceeds (after producing each directive). Additionally, for mapping any subsequent block, the latest starting point is used for the mapping.

The dimension selection operation utilizes a sequence of symbols $s_1s_2...s_n$ and produces a dimension d_i, which can be either one of the two mesh's dimensions. The logic of the dimension selection operation must be based on the sequence s_i's. For the purpose of this paper, the dimension selection operation uses the following simple functionality: the mapping dimension d_i is the horizontal if the unicode index of s_i is odd; otherwise the mapping dimension is the vertical.

Based on this, we can formally define our mapping for a symbol b_i as follows. First, the dimension selection operation uses its input symbol s_i to determine the mapping dimension d_i. The mapping operation uses the current value of the starting point and the mapping dimension d_i to map the symbol b_i and produce a directive $\pm\ x_i$. Specifically, the mapping begins from the starting point and moves along the mapping dimension toward the index of b_i in this dimension. Both the number of cells passed and the direction with respect to the reference point (starting point) are compiled into a directive –x or $+x$, where x is the distance. The mapping operation updates the starting point to the new position before mapping any new symbol b_j.

Fig. 11 provides an example. Suppose that we want to map the symbol "m" to the mesh. Suppose that the current value of the starting point is (*h* = 7, *v* = 3) and the input symbol for the dimension selection operation is "a". Since the value of the symbol "a" in the unicode coding is 97, which is odd, the mapping dimension is the horizontal dimension. Therefore, the mapping operation starts from the starting point (7, 3) and moves along the horizontal dimension to the index of "m". Since the number of passed cells is 3 and the direction of the move is to the higher indexes, the mapping yields the directive "+3" as a mapping value for "m". The mapping operation updates the starting point to the new value (10, 3). To further illustrate the mapping operation, assume now that the symbol to be mapped is "m", the starting point is (7, 3), and the input symbol to the dimension selection is "b" (instead of "a"). Since the value of "b" in the unicode is 98, which is even, the mapping dimension is the vertical. Therefore, the mapping starts from the starting point (7, 3) and moves along the vertical dimension toward the index of "m". The number of passed cells is 4 and the direction of the move is toward the higher indexes. The mapping operation produces accordingly the directive "+4" as a result. The starting point is updated to (7, 7). As a final illustration, suppose that we want to map the symbol "e", where the input of the dimension selection is "a" and the starting point is (7, 3). Since the value of "a" is 97, which is odd, the mapping dimension is the horizontal. The mapping operation hence starts from the starting point (7, 3) and moves along the horizontal dimension to the position of "e". Since the distance of the move is 5 and is toward the lower indexes, the mapping operation produces the directive "–5" as a result of the mapping. The starting point is updated to (2, 3).

VI. Mask Process

This mask process deeply alters its input block. The alteration includes (1) masking the symbols and (2) altering both the symbols and the block's structure. Fig. 12 shows the main operations of the mask process.

First, the mask process executes the *Manipulate* operation. This operation uses a sequence of the directives "$+d_1$ +

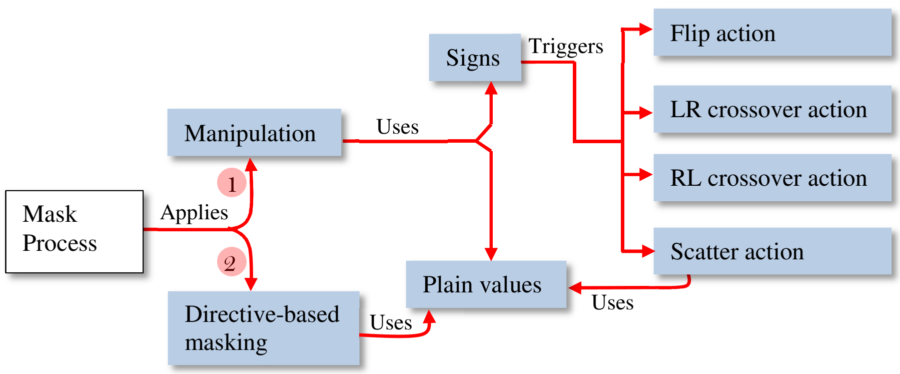
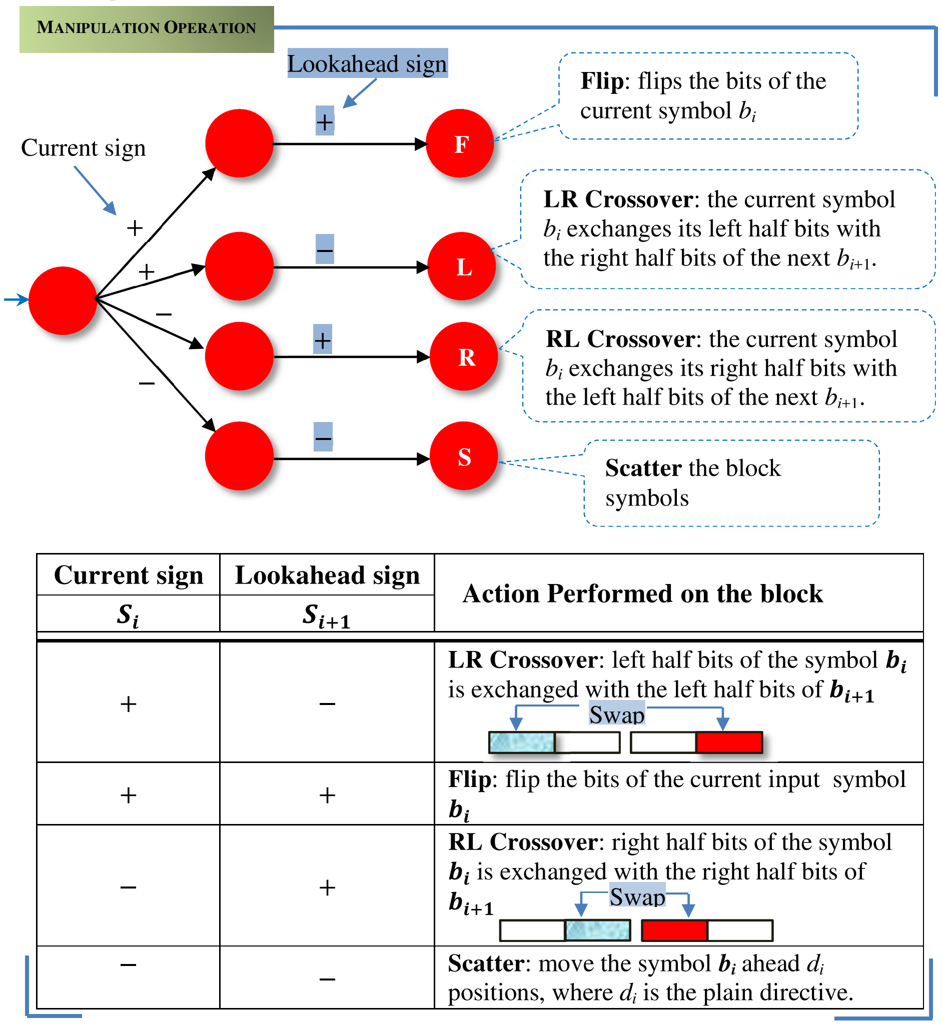


Fig. 12. the main actions of the mask operation.


MANIPULATION OPERATION
Lookahead sign
Current sign
+
+
−
−
+
−
+
−
F
L
R
S
Flip: flips the bits of the current symbol b_i
LR Crossover: the current symbol b_i exchanges its left half bits with the right half bits of the next b_{i+1}.
RL Crossover: the current symbol b_i exchanges its right half bits with the left half bits of the next b_{i+1}.
Scatter the block symbols
Swap
Swap

Current sign S_i	Lookahead sign S_{i+1}	Action Performed on the block
+	−	**LR Crossover**: left half bits of the symbol $\boldsymbol{b_i}$ is exchanged with the left half bits of $\boldsymbol{b_{i+1}}$
+	+	**Flip**: flip the bits of the current input symbol $\boldsymbol{b_i}$
−	+	**RL Crossover**: right half bits of the symbol $\boldsymbol{b_i}$ is exchanged with the right half bits of $\boldsymbol{b_{i+1}}$
−	−	**Scatter**: move the symbol $\boldsymbol{b_i}$ ahead d_i positions, where d_i is the plain directive.

Fig. 13. the logic of the manipulate operation. The figure demonstrates how the sequences of signs trigger the manipulation actions.

$d_2 - d_3$...” to make two large effects on the block’s symbols: breaking the structure of the block by reordering its symbols and altering the individual symbols by modifying or mixing some of their bits. To perform these effects, the *Manipulate* operation uses the signs of the directives to execute four different actions on the block. Fig. 12 shows these four actions: *Flip*, *LR* Crossover, *RL* Crossover, and *Scatter*. *Flip* action negates the bits of the symbol. *LR* Crossover action causes the current symbol b_i to exchange the left half of its bits with the right half bits of the next symbol b_{i+1}. *RL* Crossover action causes the current symbol b_i to exchange the right half of its bits with the left half bits of the next symbol b_{i+1}. The *Scatter* action repositions the symbol b_i in the place specified by the directive d_i.

Fig. 13 shows the conditions under which these actions are triggered. In particular, an action *X* is triggered based on a specific pattern of the current sign and the lookahead sign. Referring to Fig. 13, when the current sign is “+” and lookahead sign is also “+”, the *Flip* action is triggered and flips all the bits of the current input symbol b_i. When the current sign is “+” and the lookahead symbol is “–”, the *LR* crossover action is triggered causing the left half bits of the current symbol b_i to exchange with the right half bits of the next symbol b_{i+1}. When the current sign is “–” and the lookahead sign is “+”, the *RL* crossover action is triggered causing the right half bits of the current symbol b_i to exchange with the left half bits of the next symbol b_{i+1}. Finally, when the current sign is “–” and the lookahead sign is “–”, the *Scatter* action is triggered causing the current symbol b_i to move to the index specified by d_i.

It is clear that the former three actions (*Flip* and *LR*/*RL* Crossover) alter the individual symbols. While the latter action (*Scatter*) changes the structural relation between the symbols of the input block.

Second, the mask process introduces further masking to the block’s symbols by executing the directive-based masking operation. This operation does the masking by embedding the effects of the plain directives. In particular, the plain directives d_i’s (without the sign) are XORed with the corresponding block symbols to yield a masked block $c_1c_2...c_n$. That is c_i = $b_i \oplus d_i$ for *i* = 1, 2, ..., *n*.

We illustrate the mask process using an example. Suppose for the sake of simplicity the following 4–byte block “77, 61, 6C, 6C”. Suppose further that the sequence of directives is “+30–7–18+3”. Fig. 14 shows the steps of masking this block. The leftmost column represents the index of the current input symbol–where “0” represents is the index of the leftmost symbol in the input block. The second column shows the sequence of signs, where the currently considered sign pattern is shaded, and shows also the action performed (*LR*, *S*, etc.) based on the shaded sign pattern. The third column shows the input string (in Hex) and the input symbols which the corresponding action operates on. The rightmost column shows the sequence of directives. The manipulation process starts with the input symbol at index 0 (“77”) and the first sign pattern “+ –”. According to the logic in Fig. 13, this pattern triggers the *LR* Crossover action. The *LR* action therefore exchanges the left half bits of the current input symbol “77” with the right half bits of the next input symbol “61”, yielding the new block “17, 67, 6C, 6C”. Now, the second input symbol (at index 1) is “67” and the second sign pattern is “– –”. This sign pattern triggers the *Scatter* action, which moves the second input symbol “67” to the position specified by the plain directive of the second directive “–7”. That is, Scatter action moves “67” to the position 7 % 4 = 3. (We take the module % because the length of the input block is only four symbols.) Continuing likewise, the *Manipulate* operation produces the manipulated block “17, 6C, 66, 38”. The manipulated block is further masked by XORing the symbols with the plain part of the directives sequence. (Note the plain directives 30, 7, 18, 3 are transformed to Hex: 1E, 07, 12, 03.) The final output of the masking operation is “09, 6B, 74, 3B”.

The effect of the mask process is reversible provided that we have the directives sequence. To recover the original block, we define the inverse mask process. This process performs

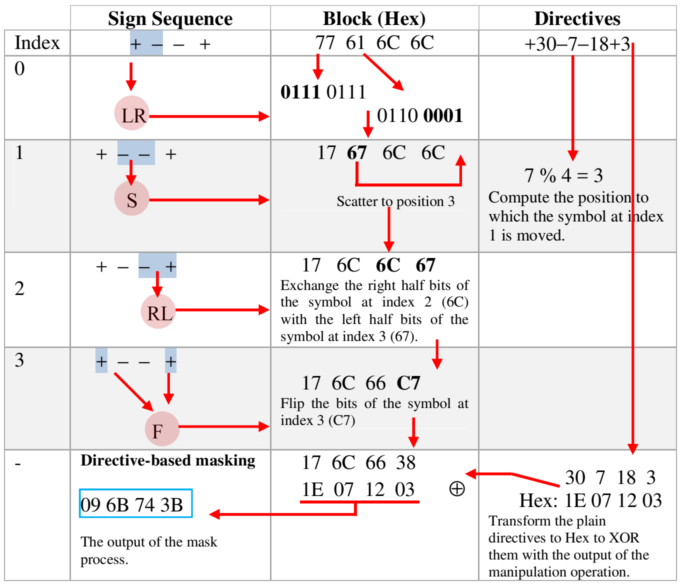

Sign Sequence
Block (Hex)
Directives
Index
+ – – +
77 61 6C 6C
+30–7–18+3
0
LR
0111 0111
0110 0001
1
+ – – +
17 67 6C 6C
S
Scatter to position 3
7 % 4 = 3
Compute the position to which the symbol at index 1 is moved.
2
+ – – +
17 6C 6C 67
RL
Exchange the right half bits of the symbol at index 2 (6C) with the left half bits of the symbol at index 3 (67).
3
+ – – +
F
17 6C 66 C7
Flip the bits of the symbol at index 3 (C7)
-
Directive-based masking
17 6C 66 38
1E 07 12 03
⊕
30 7 18 3
Hex: 1E 07 12 03
09 6B 74 3B
The output of the mask process.
Transform the plain directives to Hex to XOR them with the output of the manipulation operation.

Fig. 14. an example of the mask process.

the same operations as the mask process, but in a reverse order. That is, the inverse mask process applies first the directive-based masking operation and then the *Manipulate* operation. The directive-based masking operation applies the XOR operation to the block and the plain directives d_i’s. The manipulation operation starts from the rightmost of the sign sequence and moves backwards (instead of starting from the leftmost). For instance, if the signs sequence is “+ – – + – –”, the reverse operation starts from the rightmost pattern “– +”, – –”, “+ –”, “– +”, “– –”, and “+ –”. The *Flip*, *LR* Crossover, and *RL* Crossover actions do not change. The *Scatter* action functionality is changed. Instead of moving the symbol b_i to the position d_i % *n*, the action moves the symbol at the psition d_i % *n*, to the current index *i*.

Fig. 15 shows an example of recovering the original block “77, 61, 6C, 6C” from the masked block “09, 6B, 74, 3B”. The sequence of directives is “+30–7–18+3” (“+1E–07–12+03” in Hex.) As the figure shows, the masking process applies first the directive-based masking operation to the input block “09, 6B, 74, 3B”, yielding “17, 6C, 66, 38”. The manipulation operation is then applied starting from the last sign pattern “++” and from the end of the input block. The pattern “++” triggers the *Flip* action, which flips all of the bits of the last symbol “38”, yielding the new block “17, 6C, 66, C7”. The current symbol now is “66” (at index 2) and the sign pattern is “– +”. This pattern triggers the *RL* crossover action, which causes the right half bits of the current input symbol “66” to be swapped with the left half bits of the following symbol “C7”, yielding the new block “17, 6C, 6C, 67”. The current symbol now is “6C” (at index 1). The sign pattern is “– –”, which triggers the *Scatter* action. The symbol at the index 7 % 4 = 3, which is “67”, is moved to the current index (position 1), yielding “17, 67, 6C, 6C”. Finally, the sign pattern “+ –” triggers the *LR* crossover action, which causes the symbol at the current index 0 (“17”) to exchange the left half of its bits with the right half bits of the next symbol at index 1 (“67”), yielding “77, 61, 6C, 6C”—the original block.

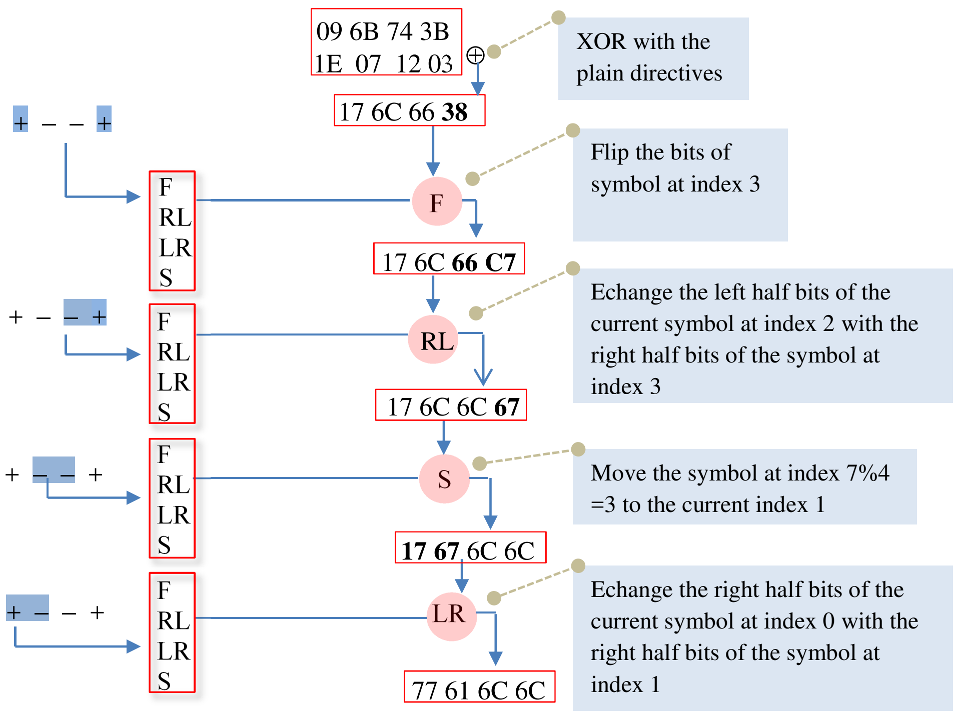

09 6B 74 3B
1E 07 12 03
XOR with the plain directives
17 6C 66 38
+ – – +
F RL LR S
F
Flip the bits of symbol at index 3
17 6C 66 C7
+ – – +
RL
Echange the left half bits of the current symbol at index 2 with the right half bits of the symbol at index 3
17 6C 6C 67
+ – – +
S
Move the symbol at index 7%4 =3 to the current index 1
17 67 6C 6C
+ – – +
LR
Echange the right half bits of the current symbol at index 0 with the right half bits of the symbol at index 1
77 61 6C 6C

Fig. 15. example of recovering the plaintext block from the masked block.

VII. The Cipher Technique

This section presents the technical details of the proposed block cipher. Subsection VII-A presents how to use the operations defined in the previous sections to encrypt blocks of plaintext. Subsection VII-B presents how to decrypt the resulting ciphertext.

A. The Encryption Process

Let B_i be plaintext block of size n and *Key* be a key of size m, where m is not necessarily equal to n. We impose no constraints on the size of the block. We also impose no specific constraints on the size of the key except those required for the key security. Therefore, a key size of 16 symbols or larger is highly recommended.

Initially, the encryption process prepares the D-Box and its inverse D-Box^{-1} as described in section II. This step is necessary for the functionality of the encryption operations and the key expansion.

After the D-Box is created, the encryption process executes as Fig. 16 shows. The process encrypts the block B_i using the *Key* in t rounds. It first expands the *Key* to size of $t \times n$ symbols, where n is the size of the plaintext block B_i and t is the number of rounds. Each round applies the diffuse and substitute operations to the input block B_i as described in sections II and IV. The outcome is passed to the mask operation. The mask operation receives also as an input a sequence of n directives obtained from the directive generator. In order for the directive generator to produce these directives, it receives two inputs: the previous plaintext block B_{i-1} and sub-key of size n symbols. The sub-key symbols are obtained from the expanded key (EK[x: y]). The directive generator maps the symbols of the previous block B_{i-1} to the mesh using the sub-key as described in Section V.

The encryption of the first block is handled slightly differently. That is because the first block B_1 has no predecessor block B_{i-1}. In this case, we use the n symbols of the *Key* instead of the block B_{i-1}. Therefore, the same n symbols of the *Key* are used as an input for both the *Mapping* and the *Dimension selection* operations.

B. The Decryption Process

The decryption process takes ciphertext and a key as an input and uses the key to produce the corresponding plaintext as an output. In order to accomplish this, the decryption process uses the inverse of the operations that are used during the encryption. The order in which the operations are applied is also reversed. Therefore, the decryption operations are executed in the following order: **Inverse Mask**, **Inverse Substitute**, and **Inverse Diffuse**. Fig. 17 shows the detailed steps of the decryption process. As the figure shows, the first operation to be applied to the input is the inverse mask operation instead of diffusion operation. The inverse substitute uses the D-Box^{-1} to reverse the effect of the substitution operation. Finally, the inverse diffuse operation is applied to reverse the effect of the diffuse operation.

In addition as Fig. 17 shows, the expanded key is used backwards. That is, the decryption process starts from the end of the expanded key (with $t \times n$ symbols) instead of starting from the beginning. Therefore, the decryption process uses the first n key symbols from the right then the second n symbols, and so on until the last left n symbols. The sequences of directives are also used backwards. As Fig. 17 shows, the last

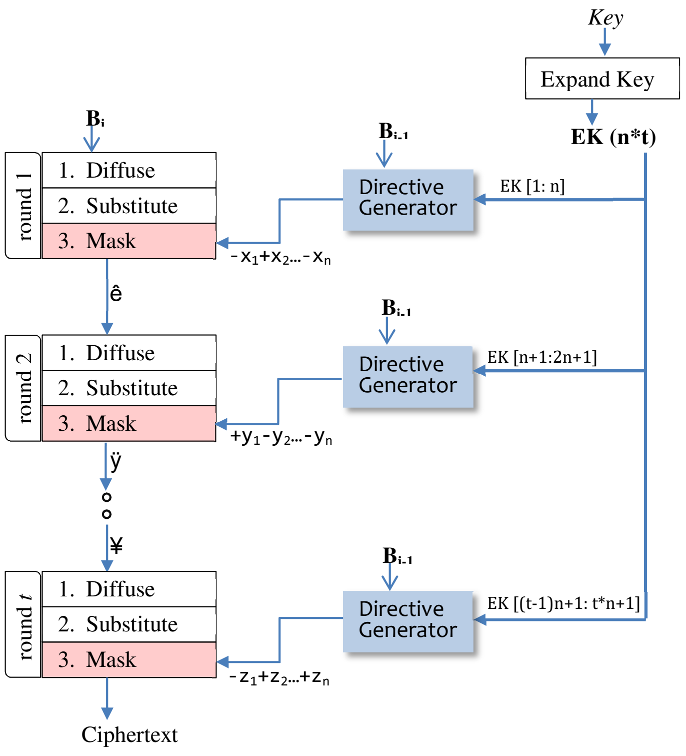


Fig. 16. the encryption process.

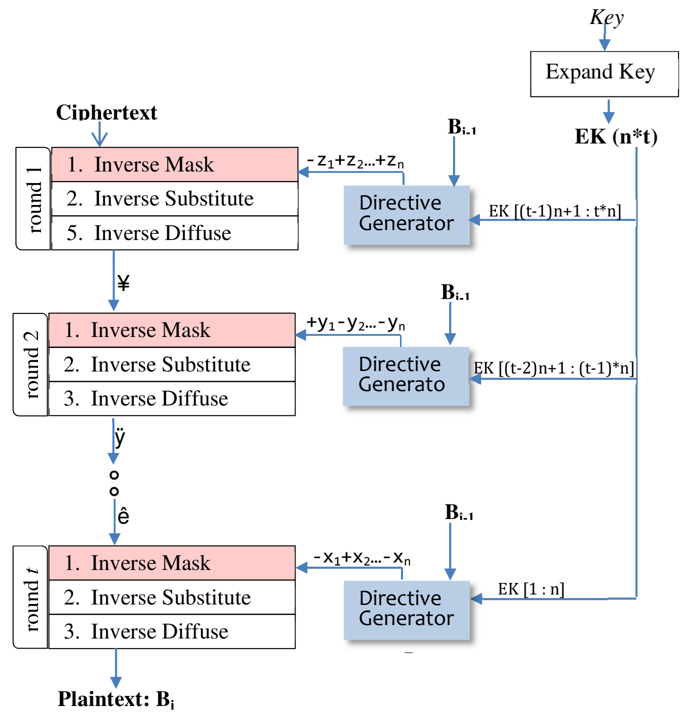


Fig. 17. the decryption process.

directive sequence in the encryption process "$-z_1 + z_2 ... + z_n$" is used first.

VIII. Related Work

The most related encryption technique to ours is the advanced encryption standard (*AES*) [3]. The *AES* encrypts a block of plaintext in many rounds. The number of rounds depends on the size of the key. In each round, *AES* applies different operations: key round, substitution using S-Box, row shifting, column mixing, and so on. The *AES* technique is NIST's standard [24]. Many researchers, however, reported major weaknesses. Specifically, the *AES* suffers from the weakness in the S-Box, making it more susceptible to computational attacks [5][19]. Several proposals have been suggested to improve the S-Box [6][20][21].These proposals claimed to improve the effectiveness of the S-Box. The effectiveness of the S-Box is further enhanced in [22]. As claimed in [22], the new enhancement increases the algebraic complexity of the S-Box and therefore it becomes more immune against differential and algebraic attacks. These extensions, however, have their inherent drawbacks as pointed out in [5].

Despite the importance of these improvements and the work around methods, we believe that strengthening the substitution part of the encryption algorithms (Mainly *AES*) cannot be properly done by merely adding more mathematical transformations. We think the strength of the substitution must be based on the key as proposed by the technique herein. In other words, instead of having static substitution table (S-Box), it is more effective to have a more dynamic substitution operation whose state must depend on the key and change according to the changes of the this key (D-Box).

Data encryption standard (DES) and its successive improvements such as Triple DES [9][10] are symmetric block ciphers that follow Feistel cipher structure [30]. Their encryption model depends on applying a set of substitutions and manipulation operations augmented with key evolution. Although this algorithm is used to be a standard, its security challenged [23][25]. Even with fundamental improvement such as using two keys, the algorithm still suffers and its security in question [23].

MARS, Blowfish, and Serpent [14][11][13] are symmetric block ciphers. Their encryption model depends like [9] on a set of substitutions and manipulation operations. Although their authors purport that these methods provide high levels of security, they failed to pass important randomness tests [27].

LEX encryption technique [16] is a stream cipher that is inspired by the one-pad encryption method (called also Vernam cipher). Although *LEX* claimed to have strong security properties, it uses the same round key repeatedly. This repeated use of the key makes it vulnerable to key-recovery attacks [28]. Our technique never uses the same key; each round uses a different version of the key and thus each block is encrypted using different keys.

Camellia encryption method [17] is a symmetric encryption technique, supporting128-bit block size and 128, 192, 256-bit key. Efficiency on both software and hardware platforms is a remarkable characteristic of Camellia in addition to its high level of security. The main important difference between our technique and Camellia is that ours has more effective diffusion operation along with a dynamic substitution box.

To conclude this section, it is worth mentioning that our proposed encryption method possesses several important characteristics that make it unique. First, in contrast to all encryption methods, our technique implicitly uses the key in every encryption operation. Second, our technique uses the plaintext to introduce further confusion to the encryption. In particular, it embeds the effect of the previous plaintext block in the encryption of the next block. As such, changes in the previous block impact not only the encryption of the previous block itself but the encryption of the next block as well; a feature that–to the best of our knowledge–is unique to our cipher. Third, our technique triggers the manipulation actions (these actions belong to the masking process) based on the signs of the directives. Since the signs of the directives depend on both the key and the plaintext blocks, the sequence of the triggering highly relies on the plaintext blocks and keys and greatly sensitive to their changes. This makes predicting the triggering order is infeasible. Finally, the functionality of our substitution operation changes with the changes of the key. This adds additional confusion layer, making the substitution step much more effective than the static substitution used by other encryption techniques.

IX. Performance Analysis

We present our performance analysis in this section. We first present an example that demonstrates some of our technique's features and then analytically discuss its security properties in Subsection IX-B. We present the empirical evaluation in Subsection IX-C.

A. Encryption Example

We start our analysis by presenting examples of the technique's output (ciphertext). The examples are meant to be simple but indicative. For the sake of simplicity, we assume that the plaintext blocks and the keys are of length 16 symbols. Fig. 18 shows the ciphertexts for the corresponding plaintexts, keys, and previous blocks. Referring to the figure, one can see that the output changes drastically when the input slightly changes. Consider for instance the first five rows, which are encrypted using the same key. Although these plaintexts differ in a single bit, their respective ciphertexts are greatly different. Changing a single bit in the key causes the ciphertexts for the same plaintexts to be really different (e.g. compare between the ciphertexts in rows 6 and 7). Finally, changing the previous block highly impacts the resulting ciphertexts. For instance, a quick look at rows 1 and 9 shows that changing a single bit in the previous block causes large changes to the ciphertexts of the same plaintext block.

This property of the proposed technique is very important. From one hand, a tiny change to any of the input (plaintext, key) results in extremely large changes to the respective ciphertexts. From the other hand, a tiny change to the previous plaintext block also largely impacts the ciphertext of the following plaintext block. From the security prospective, the proposed technique causes the relationship between plaintext and ciphertext to be so complicated and untraceable.

B. Analytical Performance Evaluation

The proposed technique has high confusion. The key is never explicitly used in the encryption; it is implicitly used via the directives. The key "trace" in the ciphertext is therefore so small to help predicting the used key. As such, the relationship between the key and the ciphertext is untraceable.

The proposed technique is highly sensitive to the changes of the input (see Fig. 18). This sensitivity provides an additional strong security guard. It makes the relationship between a key, ciphertext, and plaintext so complicated in a sense that if any of them changes, the respective ciphertext greatly changes. We attribute this high sensitivity to the changes of the input (plaintext or a key) to all the encryption operations (especially the diffuse and key expansion operations).

The state of the substitution box (D-Box) highly depends on the encryption key. Each different key results in a largely different new state and consequently results in different substitution outcomes. This means that changes of the key cause the same plaintext block to be substituted differently. This is in contract to the other encryption algorithms (especially *AES*), where the substitution of a block is independent of the key. That is, the outcome of substituting a block remains the same regardless of the used key.

The key expansion operation adopts a highly complicated computational model. From one hand, the substitution sub-operation relates the process of substituting the current symbol to the outcome of substituting the proceeding symbols. This means that the outcome of the substitution for the current symbol k_i is impacted by all the previous symbols k_j (*j*=1, 2, ..., *i*-1). On the other hand, the key expansion manipulation sub-operation is highly complicated process. It partially depends on the order of the key's symbols. For each new key, we have a new state (symbol order). In addition, the selection of the key's manipulation operations depends on a random process. In other words, although the order of the symbols may bias the selection toward the operation with the higher ratio, the selection of the operation to be applied is nevertheless random. Even more, the random process is always seeded with latest version of the key. These properties assure that the key manipulation operations have a greatly complicated functional behavior in a sense that the number of states is tremendously large.

The mask process has perhaps the highest impact on its input block. It makes deep changes to both individual symbols and to the structure of the block, largely diverging the resulting block from the input block. The functionality of this operation fully depends on directive sequences. The manipulation operation, which modifies the individual symbols and the structure of the block, depends on the pattern of the signs (of the directives). Since, this pattern depends on both the key and the previous block, changes to either the key or the previous block certainly results in a different pattern (regardless of the magnitude of the change) and consequently in a different functional behavior. Further the directive-based masking operation, which embeds the effect of the plain directive (value without the sign), depends also on the key and the previous block and changes according to their changes. As such, a change to the key or the previous block creates different modifications to the output of the masking operation.

Previous block: 0000000000000000		
Key	**Plaintext**	**Ciphertext**
0000000000000000	0000000000000000	59 e0 5a 42 96 f4 13 6d 46 c9 c3 c7 5d 27 70 57
0000000000000000	**1**000000000000000	a6 9d c4 12 55 c2 57 da 04 ba b5 62 f4 49 b0 cb
0000000000000000	000000000000000**1**	8e a2 13 87 55 d7 8f dc 13 ab 66 c1 97 76 65 4a
0000000000000000	000000000**1**000000	67 44 cc ec e5 e4 3f 2b f8 84 9f ac 11 30 25 8a
0000000000000000	**1**00000000**1**000000	f8 29 21 d8 33 49 2f f0 3b 57 74 e1 20 08 a3 32
1000000000000000	0000000000000000	44 28 e3 80 4e 0b fb 20 98 70 1b d2 59 4b 57 fe
000000000000000**1**	0000000000000000	6d 42 3a aa ae c0 d8 91 4c 48 d6 65 d3 31 cb 22
Previous block: **1**000000000000000		
0000000000000000	0000000000000000	9f 66 a2 3e 34 f5 d4 96 ee 40 5d 92 89 ca c3 5c
Previous block: 000000000000000**1**		
0000000000000000	0000000000000000	71 32 c7 77 84 96 19 9f 12 3e 82 5b 55 77 fe 76
Previous block: 0000000000000001		
ACF98IFTRmk90AGT	To beornotto be!	5b 8c 80 61 77 cc 3a 81 b7 41 3c 39 6a 60 35 d8
ACF98IF**S**Rmk90AGT	To beornotto be!	f3 99 d0 49 68 b8 4e 4c 7a 88 a8 e8 d2 7d 25 ae
ACF98IFSRmk90AGT	To beornotto be**.**	1b db 47 a4 3f 76 03 50 b3 7 95 0e 19 1b d0 9f

Fig. 18. The proposed technique output examples.

C. Empirical Evaluation

We tested the performance our encryption technique according to the testing rules specified by the national institute for standards and technology–NIST [27]. We specifically prepared the testing data as specified in [26].

1) Randomness Statistical Tests: We used the following tests to evaluate the randomness properties of our technique [27].

- *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.

2) Randomness Hypotheses: We have two hypotheses to test:

- H_0 (*Null*): the output of the encryption technique is random.
- H_1 (*Alternative*): the output of the encryption technique 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).

3) Test Data: The testing data were prepared according to [26]. Without losing the generality, we confine the D-Box to be 16×16. This allows for representing each symbol by 8 bits. We used the following sets of data to test our encryption technique.

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

Firstly, to study the sensitively of our algorithm to the key change, we created and analyzed 456 sequences of size 32,768 bits each. We used a 256-bit (32 bytes) plaintext of all zeros and 456 random keys each of size 128 bits (16 bytes). Each sequence was created by concatenating 128 derived blocks constructed as follows. Each derived block is created by XORing the ciphertext created using the fixed plaintext and the 128-bit key with the ciphertext created using the fixed plaintext and the perturbed random 128-bit key with the i^{th} bit changed, for $1 \leq i \leq 128$.

Secondly, to analyze the sensitivity to the plaintext changes, we created and analyzed 456 sequences of size 32,768 bits each. We used 456 random plaintexts of size 256 bits (32 bytes) 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 256-bit plaintext with the ciphertext created using the 128-bit key and the perturbed random 256-bit plaintext with the i^{th} bit changed, for $1 \leq i \leq 256$.

Thirdly, to study the correlation of plaintext–ciphertext pairs, we constructed 456 sequences of size 115,712 bits per a sequence. Each sequence is created as follows. Given a random 128-bit key and 452 random plaintext blocks (the

TABLE II. Key Avalanche Test Results

Randomness Test	Successes	Failures	Success Rate	Upper limit of CI (0.05)
Runs test	452	4	99.1%	36.76
Monobit test	445	11	97.6%	36.76
Spectral test	424	32	93.0%	36.76

TABLE III. Plaintext Avalanche Test Results

Randomness Test	Successes	Failures	Success Rate	Upper limit of CI (0.05)
Runs test	448	9	98.2%	36.76
Monobit test	442	14	96.9%	36.766
Spectral test	418	**38**	91.7%	36.76

TABLE IV. Plaintext/Ciphertext Correlation Test Results

Randomness Test	Successes	Failures	Success Rate	Upper limit of CI (0.05)
Runs test	447	9	98.0%	36.76
Monobit test	443	13	97.1%	36.76
Spectral test	421	**35**	92.3%	36.76

block's size is 256 bits), a binary sequence was constructed by concatenating 452 derived blocks. A derived block is created by XORing the plaintext block and its corresponding ciphertext block. Using the 452 (previously selected) plaintext blocks, the process is repeated 455 times (one time for every additional 128-bit key).

During the encryption, we set the number of rounds t to 8 rounds for each block because this number gives high performance. (Please see appendix A for more detailed discussion on the number of rounds t.)

Tables II, III, and IV show the results of the applied randomness test to the above data sets. The tables show the applied tests, the number of sequences that passed the respective test, the number of failed sequences, and the rate of success. For each test, the significance level was fixed at 0.05, which implies that, ideally, no more than five out of hundred binary sequences may fail the corresponding test. However, in all likelihood, any given data set will deviate from this ideal case. A more realistic interpretation is to use a confidence interval (CI) for the proportion of binary sequences that may fail at the 0.05. The rightmost column shows the maximum number of binary sequences that are expected to fail the corresponding test. For instance, a maximum of 36.76 (or 36) binary sequences are expected to fail each of the three tests.[2]

All the three tables show that the success rate is remarkably high. In fact, 99.1% of the sequences for testing the key avalanche (Table II) passed the Runs test, 97.6% passed the Monobit test, and 93.0% of the sequences passed the Spectral test. Tables III and IV lead to the same conclusions. Additionally, the rightmost column, which specifies the highest number of sequences that are expected to fail the test, shows that number of sequences that failed the corresponding randomness tests is fewer than the expected number at significance level of 0.05. Table III shows, however, that the number of the sequences that failed Spectral test is actually greater than the highest number of sequences that are expected to fail (actual 38, expected 36). Also, in Table IV, the number of the sequences that failed Spectral test is 35, which is very close the maximum number of the sequences that are expected to fail.

[2]The maximum number of binary sequences that are expected to fail at the level of significance α is computed using the following formula [29]: $S(\alpha + 3 \times \sqrt{\frac{\alpha\times(1-\alpha)}{S}})$, where S is the total number of sequences and α is the level of significance.

X. Conclusions and Future Work

We proposed a full-fledged encryption technique. This technique uses effective operations to manipulate blocks of plaintext and create deep changes to both the structure of the block and the individual symbols. By virtue of these operations, the technique ensures high diffusion, confusion, and avalanche effect. The performance numbers in subsection 9.3 clearly indicate the high avalanche effect. The technique has passed three NIST–recommended randomness tests for evaluating the effeteness of encryption technique. In fact, the least percentage of the sequences that passed these tests was 91.7%.

We have two main objectives for future work. First, we want to replace the built-in random generator, which is used in the key expansion operation, with a more effective random generator. Second, we plan also to apply more randomness tests on a larger set of plaintexts and keys.

References

[1] M. J. Al-Muhammed and R. Abuzitar, κ–Lookback Random-Based Text Encryption Technique, *Journal of King Saud University-Computer and Information Sciences*, 2017. https://doi.org/10.1016/j.jksuci.2017.10.002

[2] M. J. Al-Muhammed and R. Abuzitar, Dynamic Text Encryption, *International Journal of Security and its Applications* (*IJSIA*), Vol. 11, No. 11, pp. 13–30, November 2017.

[3] J. Daemen and V. Rijmen, The Design of RIJNDAEL: *AESThe Advanced Encryption Standard*, Springer, Berlin, Germany, 2002.

[4] A. Abusukhon, Z. Mohammad, and M.Talib, A Novel Network Security Algorithm based on Encryption Text into a White-Page Image, Proceedings of the World Congress on Engineering and Computer Science, Vol. I WCECS 2016, October 19-21, 2016, San Francisco, USA.

[5] N. A. Azam, A Novel Fuzzy Encryption Technique Based on Multiple Right Translated AES Gray S-Boxes and Phase Embedding, *Security and Communication Networks*, Vol. 2017, 9 pages. https://doi.org/10.1155/2017/5790189

[6] J. Liu, B. Wei, X. Cheng, and X. Wang, An AES S-box to Increase Complexity and Cryptographic Analysis. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications* (*AINA*'05), pp. 724–728, Taipei, Taiwan, March 2005.

[7] D. Nilesh and M/ Nagle, The New Cryptography Algorithm with High Throughput, 2014 *International Conference on Computer Communication and Informatics* (2014), October 2014.

[8] B. Modi and V. Gupta, A Novel Security Mechanism in Symmetric Cryptography Using MRGA. In: Sa P., Sahoo M., Murugappan M., Wu Y., Majhi B. (eds) *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications. Advances in Intelligent Systems and Computing*, vol 719. Springer, Singapore, 2018

[9] S.-J. Han, the Improved Data Encryption Standard (DES) Algorithm. In *Proceedings of IEEE* 4^{th} *International Symposium on Spread Spectrum Techniques and Applications*, pp. 1310–1314, Germany, 1996.

[10] J. Verma and S. Prasad, Security Enhancement in Data Encryption Standard. In: Prasad S.K., Routray S., Khurana R., Sahni S. (eds) *Information Systems, Technology and Management* (*ICISTM*) 2009. Communications in Computer and Information Science, Vol. 31. Springer, Berlin, Heidelberg, 2009.

[11] T. Nie and T. Zhang, A Study of DES and Blowfish Encryption Algorithm, In P*roceedings of IEEE Region* 10^{th} *Conference*, Singapore, Jan. 2009.

[12] P. Patil, P. Narayankar, D.G Narayan, S. M. Meena, A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish, Procedia Computer Science, Vol. 78, pp. 617-624, 2016.

[13] R. Anderson, E. Biham and L. Knudsen: Serpent: A Proposal for the Advanced Encryption Standard. Available at http://www.cl.cam.ac.uk/ rja14/Papers/serpent.pdf

[14] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, and N. Zunic, The MARS Encryption Algorithm, IBM, August 1999.

[15] S. A. Y. Hunn, S. Z. binti . Naziri, and N. B. Idris, The Development of Tiny Encryption Algorithm (TEA) Crypto-core for Mobile Systems, 2012 *IEEE International Conference on Electronics Design, Systems and Applications* (*ICEDSA*), Kuala Lumpur, Malaysia, Nov. 2012.

[16] A. Biryukov. Design of a new stream cipherLEX. In M. Robshaw and O. Billet (Eds.): New Stream Cipher Designs, LNCS 4986, Springer–Verlag Berlin Heidelberg, pp. 48–56, 2008.

[17] K. Aoki, T. Ichikawa , M. Kanda , M. Matsui , S. Moriai , J. Nakajima , T. Tokita, Camellia: A 128–Bit Block Cipher Suitable for Multiple Platforms-Design and Analysis, In *Proceedings of the* 7^{th} *Annual International Workshop on Selected Areas in Cryptography*, pp.39–56, August 14–15, 2000.

[18] M. F. Mushtaq, S. Jamel, A. H. Disina, Z. A. Pindar, N. S. A. Shakir, and M. M. Deris, A Survey on the Cryptographic Encryption Algorithms, International Journal of Advanced Computer Science and Applications (IJACSA) , Vol. 8, No. 11, pp. 333–344, 2017.

[19] J. Rosenthal, A Polynomial Description of the Rijndael Advanced Encryption Standard, J*ournal of Algebra and Its Applications*, Vol. 2, No. 2, pp. 223–236, 2003.

[20] L. Jingmei, W. Baodian, and W. Xinmei, One AES S-box to Increase Complexity and its Cryptanalysis, *Journal of Systems Engineering and Electronics*, Vol. 18, No. 2, pp. 427–433, 2007.

[21] L.Cui and Y. Cao, A New S-box Structure Named Affine-Power Affine, *International Journal of Innovative Computing, Information and Control*, Vol. 3, No. 3, pp. 751–759, 2007.

[22] M. Khan and N. A. Azam, Right Translated AES Gray S-boxes, *Security and Communication Networks*, Vol. 8, No. 9, pp. 1627–1635, 2015.

[23] C. J. Mitchell, On the Security of 2–Key Triple DES, *IEEE Transactions on Information Theory*, Vol. 62, No. 11, pp. 6260–6267, Nov. 2016.

[24] NIST Special Publication 800–67 Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher Revision 1, Gaithersburg, MD, USA, Jan. 2012.

[25] S. Lucks, Attacking Triple Encryption, In *Proceedings of the* 5^{th} *International Workshop Fast Software Encryption* (*FSE*), pp. 239–253, Mar. 1998.

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

[27] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. 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.

[28] A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser, ALE: AES-Based Lightweight Authenticated Encryption. In: S. Moriai (ed.) FSE 2013. LNCS, Vol. 8424, pp. 447–466. Springer, Heidelberg, 2014.

[29] J. Soto, Randomness Testing of the Advanced Encryption Standard Candidate Algorithms, NIST IR 6390, September 1999.

[30] W. Stallings, *Cryptography and Network Security*: *Principles and Practice*, 7^{th} edition, Pearson, 2016.

Appendix A
Additional Data

A. Key-Based Numbers

We describe in this section how we generate the integers I_i for reordering the content of D-Box. The process involves two steps.

*1) **STEP I**:* In this step, the encryption key is processed using the key handling procedure proposed by *AES* (advanced encryption standard). The *AES* key handling algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (176 bytes). The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word *w*[*i*] depends on the immediately preceding word, *w*[*i*-1], and the word four positions back, *w*[*i*-4]. In three out of four cases, a simple XOR is used. For a word whose position in the w array is a multiple of 4, a more complex function is used. Fig. 19 illustrates the generation of the first eight words of the expanded key, using the symbol *g* to represent the complex function.

The function *g* consists of the following sub-functions:

1) **RotWord** performs a one-byte circular left shift on a word. This means that an input word $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.
2) **SubWord** performs a byte substitution on each byte of its input word, using the static substitution box (or Sbox)—See Fig. 20.
3) The result of steps 1 and 2 is XORed with a round constant, Rcon [*j*].

The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as:

```
Rcon[j] = (RC[j],0,0,0),
where RC[j] = 2 . RC[j-1] and RC[1] = 1
```

The multiplication is defined over the field *GF*(2^8). The values of RC[j] in hexadecimal are defined in Table V.

*2) **STEP II**:* In this step, the last 32 bytes (out of 176) of the sequence generated by Step I are used as a seed for the key-based number generator, which is described in [1]. Fig. 21 shows the algorithmic steps of the generator.

The key-based numbers are generated using the steps (2)–(5). As the figure shows, the symbols of the seed are summed

TABLE V. THE VALUES OF RC[J]

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

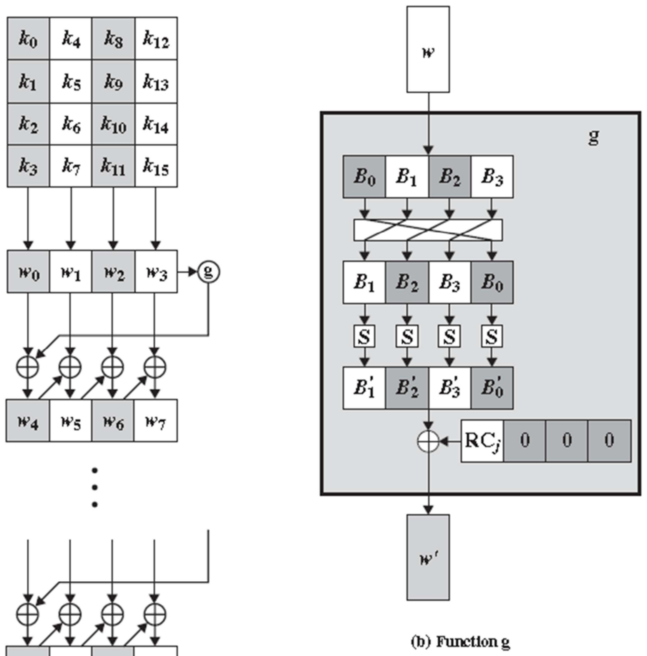

k_0 k_4 k_8 k_{12}
k_1 k_5 k_9 k_{13}
k_2 k_6 k_{10} k_{14}
k_3 k_7 k_{11} k_{15}
w_0 w_1 w_2 w_3
g
w_4 w_5 w_6 w_7
w_{44} w_{45} w_{46} w_{47}
w
g
B_0 B_1 B_2 B_3
B_1 B_2 B_3 B_0
S S S S
B'_1 B'_2 B'_3 B'_0
RC_j 0 0 0
w'
(b) Function g

Fig. 19. the AES key handling procedure.

by multiplying the integer value of each seed's symbol k_i by its position in the seed (step 2). The sum is then circularly left shifted *n* positions to yield *Sh* (*n* is the value of the first two symbols in the seed) in step 3 and *Sh* is XORed with the Sum in step 4 to yield *Ls*.

Assuming the number of symbols in the D-Box is *N*, the number *Ls* is therefore adjusted by taking the module *N*. The seed is updated in step (5) by concatenating the seed with the current sum. The seed hence grows after each iteration. If, at any iteration, step (2) results in overflow in the sum, the procedure reduces the seed to 32 symbols by using the middle 32 unicode symbols as a new value for the seed. Steps (2) through (5) repeat until the condition no longer holds. (The condition determines the desired number of values to be generated.)

It is worth mentioning that the numbers created by the generator in Fig. 21 are random. The proof of the randomness properties of the generator in Fig. 21 is beyond the scope of this paper and can be found elsewhere [1].

B. Number of Rounds (t)

The number of the encryption rounds *t*, applied to each plaintext block, is very important and has a really large impact on the randomness properties of the output (ciphertext). More rounds (large values of *t*) result in higher diffusion and confusion (random output). More rounds, however, increase the time for encrypting a block. Thus, we look for a value for *t* that achieves two objectives: (1) high randomness in the output and (2) short execution time.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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

Fig. 20. the AES SBox.

```
    (1) $Seed = key$
  Repeat
    (2) $Sum = \sum_{i=1}^{|seed|} i \times k_i$
    (3) $Sh = LShift(sum, n)$
    (4) $Ls = (Sum \oplus Sh) \% N$
    (5) $Seed = concatenate(Seed, Sum)$
Until condition
```

Fig. 21. the algorithmic steps for generating key-based numbers.

To find the best number of rounds, we conducted many experiments, where we gradually increased the number of rounds from 1 to 10. We used in our experiments the same plaintexts in Section IX. For each value of *t*, we applied our encryption technique to 200 sequences each is of size 32,768 bits. We then subjected the resulting ciphertexts to three randomness tests (Runs, Monobit, and Spectral tests).

Table VI shows the results of the experiments. The results are presented in terms number of rounds *t*, the number of the sequences that passed the corresponding randomness test, and the rate of success. As the numbers show, there is a significant improvement in the success rate as *t* increases from 1 to 8. When *t* is greater than 8, the improvement in the success rate becomes slightly small. For instance, increasing the number of rounds from 8 to 9 slightly improves the rate of success (especially for Spectral), but this slight improvement causes really large increase in the time (see Fig. 22).

Fig. 22 plots the time in milliseconds as a function of the number of rounds. The time linearly increases as the number of rounds increases. This increase in the time appears to be significant. Considering both Table VI and Fig. 22, one can conclude that the amount of the improvement in the rate of success justifies the time increase up to *t*= 8. When *t* is larger than 8, the improvement in the success rate does not really justify the incurred time overhead.[3]

[3]The purpose of presenting the time required for encrypting each sequence is not to show the time performance of our technique. The implementation is only a proof of concept and we did not optimize for time performance. We only mention the time to assert that increasing the number of the rounds increases the success rate, but this increase also entails larger time overhead.

TABLE VI. CHANGES IN THE RANDOMNESS RATE AS THE NUMBER ROUNDS INCREASES

No. of Sequences	Rounds (t)	Test	Successes	Success Rate (%)
200	1	Runs Test	100	50%
		Monobit	25	22.5%
		Spectral	0	0%
200	2	Runs Test	119	59.5%
		Monobit	97	48.5%
		Spectral	1	0.5%
200	3	Runs Test	122	61%
		Monobit	96	48%
		Spectral	3	1.5%
200	4	Runs Test	156	78%
		Monobit	133	66.5%
		Spectral	3	1.5%
200	5	Runs Test	183	91.5%
		Monobit	167	83.5%
		Spectral	58	29%
200	6	Runs Test	188	94%
		Monobit	171	85.5%
		Spectral	103	51.5%
200	7	Runs Test	192	96%
		Monobit	182	91%
		Spectral	147	73.5%
200	8	Runs Test	199	99.5%
		Monobit	195	97.5%
		Spectral	186	93%
200	9	Runs Test	200	100%
		Monobit	195	97.5%
		Spectral	189	94.5%
200	10	Runs Test	200	100%
		Monobit	197	98.5%
		Spectral	188	94%

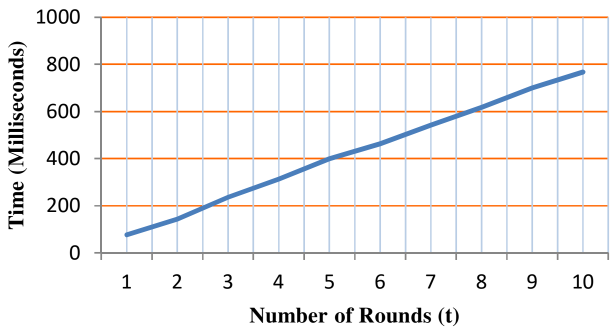


Fig. 22. The time increase as a function of the number of rounds.