International Journal of Security and Its Applications
Vol. 11, No. 11 (2017), pp.13-30
http://dx.doi.org/10.14257/ijsia.2017.11.11.02

Dynamic Text Encryption

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

[1,2] Faculty of Information Technology
American University of Madaba, Madaba, Jordan
[1]m.almuhammed@aum.edu.jo [2]r.abuzitar@aum.edu.jo

Abstract

Although many encryption algorithms achieve high levels of security, they adopt a static computational behavior that is ensured only by the secret key. If the key is predicted, the ciphered text can be easily decrypted, and thus it is unclear how the security of these algorithms will play out given the tremendous increase in processor power and the availability of sophisticated tools that can analyze ciphered text and predict a key. This paper proposes a dynamic approach to encryption in which a static knowledge of symbols and mapping tools is defined, but also in which the computations of the algorithm, and consequently the encryption, is dynamically controlled by adjusting the behavior of the operations. In addition to a key, the dynamic behavior of the algorithm provides additional tools such as masking and mapping rules to protect ciphered. Experiments conducted with our prototype implementation showed that our system is effective with linear execution time and fixed space requirements.

Keywords: *Dynamic Text Encryption; Dynamic Process; Mapping Rules; Encryption Masking; Dynamic Security Control*

1. Introduction

Securing information is, and will remain, an important goal. Many encryption algorithms in literature such as [16, 17], and others [12, 13, 1, 4, 6, 3, 2, 9, 5, 7, 8, 18, 19, 20] can achieve high levels of security. However, they adopt a static computational behavior because their security solely depends on the key, not on computations. AES and DES do go a step farther by padding random bits to the message with the purpose of varying the ciphered text even for the same plain text. Although, this adds some dynamism to the encryption, it is still added data rather than an additional dynamic process whose behavior can be more readily changed.

To produce effective tools for encryption, algorithms with true dynamic behavior are required. We believe that the true dynamic behavior can be better achieved using dynamic processes whose behavior is controlled through operations. It is the dynamic algorithms that can make available other tools to boost the security of the encryption. Intuitively, when the algorithm depends on more than one factor for securing information, breaking one factor will not be sufficient to decrypt.

To this end, our paper proposes a different approach to encryption. This approach depends on the ability to change the way the encryption is computed in addition to a key. Instead of padding random bits to the message to vary the ciphered text, our approach uses a dynamic process that consists of operations and mapping rules. The dynamic process enables the algorithm to change the way it encrypts plain text by appropriate changes to the operations and the mapping rules. With this dynamic behavior, a key still has a role in the security, but acquiring the key is inadequate to decrypt. Decryption

Received (July 15, 2017), Review Result (November 6, 2017), Accepted (November 7, 2017)
* Corresponding Author

ISSN: 1738-9976 IJSIA
Copyright ⓒ 2017 SERSC Australia

requires additional knowledge of the execution pattern that has been used to encrypt the text.

Our algorithm encrypts each symbol in plain text as directives by mapping symbols into two-dimension space. Figure 1 shows an example of plain text and its encryption in terms of directives. As the figure shows, each directive consists of (1) an integer that represents the distance of the move within the space and (2) a direction, designated as "+" or "–", within the space.

To be or Not to be.
-10+17-3-6+10-7+5-12-6+13+10-5-2-13+13+9-12-3+10

Figure 1. A Plain Text and the Directives that Encrypt This Text

Our dynamic approach to security has the interesting advantage of being fully declarative. The algorithm that performs the encryption is fixed. As a consequence, to control the level of the security, it is only necessary to update only the operations.

The paper makes the following contributions. First, it proposes a dynamic approach to encryption. This approach enables changing security levels by adjusting the behavior of the operations rather than recoding the algorithm per se. Second, it provides additional unique factors to strengthen the security of the encryption in addition to a key. Third, the algorithm has a linear time complexity and a fixed storage requirement, and both complexities are independent of any changes to the security levels. Forth, the algorithm can be easily parallelized.

We present our contributions as follows. Section 2 introduces and gives the necessary details about the main components of the algorithm. Section 3 shows the general architecture of the system. Section 4 discusses the encryption process. Section 5 discusses the decryption process. Section 6 presents the parallelized version of the algorithm. Section 7 provides performance analysis for our approach. In Section 8 we give concluding remarks and directions for future work.

2. Components Relationship

Our algorithm is composed of several components that work synergistically to encrypt/decrypt texts. Figure 2 shows the relationship between the components in the notation of [14].

The components define the necessary data and behavior for delivering the desired functionality. The component *Program* is central since it represents the program, which in turn is composed of a set of Directives. Each directive consists of a Direction and Distance (as indicated by the black-filled triangle).

The components are connected through functional relationships. The relationship "is generated by" defines a functional relationship between the *Program* and *Generator*, which means that *Program* is created using some *Generator*.

Each component has associated operations that describe its functionality. Figure 3 shows sample of operations for several components in Figure 2.

The operations manipulate a component's data. For example, the operation *rotateLeft*(*distance*, *key*) manipulates the data of the *Distance* by left rotating the distance a number of positions depending on a criterion that is based on a key. In addition, because the components define the necessary functionality for encryption and decryption, we associate with every operation that performs masking or encryption another operation that unmasks or decrypts. Referring to Figure 3, we associate with the operation *rotateLeft*(*distance*, *key*), which left rotates distance, the operation *rotateRight*(.,.) to reverse its functionality. Also, we associate with the operation *shiftForward*(.,.) the operation *shiftBackward*(.,.) to reverse its functionality.

 Copyright ⓒ 2017 SERSC Australia

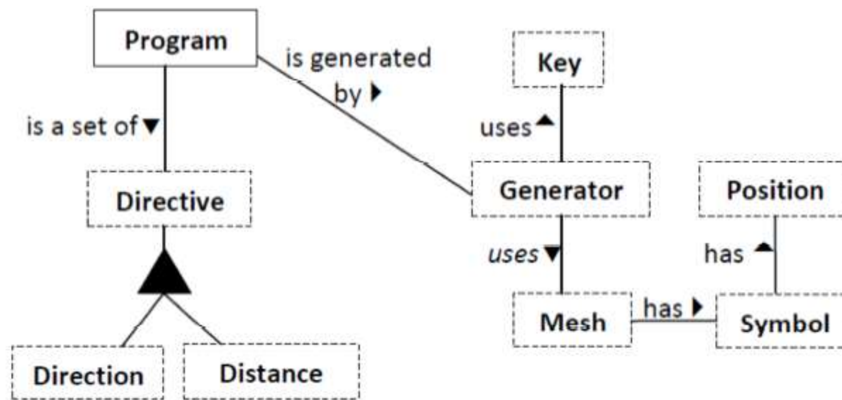

Figure 2. The Major Components for Encryption and Decryption Algorithm

```
Generator
  encrypt(plain text)  //encrypt the plain text
  decrypt(ciphered text)  //decrypt the ciphered text
  ...
Mesh
  //reorganize the symbols in a dimension of the Mesh based on the key.
   Order(dimension, key) returns reorganized symbols of dimension.
  //select the dimension of the move to be along horizontal/vertical dimension based on criterion.
   selectDimensionDistanceBased(distance)returns dimension ∈{Horizontal,
    Vertical}
   selectDimensionKeyBased(key)returns dimension ∈{Horizontal, Vertical}

    ...
Direction
  Internal representation: Symbol ∈{+, -, U}
  directionWRTPoint(Position) returns direction
 //mask and unmask the direction with respect to a reference point based on key.
  maskDirectionWRTPoint(key) returns direction
  unmaskDirectionWRTPoint(key) returns direction
   ...
Position
   Internal representation: non-negative integer

   ...
 Distance
   DistID //identifier
   Internal representation: non-negative integer
     //compute the distance between p1 and p2.
   distanceBetween(p1:Position,p2:Position) returns abs(p1-p2)
   // left /right rotate the magnitude by n positions.
   rotateLeft(distance, key)returns left-rotated distance by the value of
                         key-digit.
   rotateRight(distance, key) returns right-rotated distance by the value of
                         key-digit

   //shift forward/backward the distance based on currently read key digit. They oppose each other
  shiftForward(distance, key)returns shifted distance by key value
  shiftBackward(distance, key)returns shifted distance by key value
Symbol
  Internal representation: Unicode symbols
Key
   //expand the key by applying mutation and crossover operations on the key
   expand(key)returns expanded key.
   //creates an initial reference point from the key.
   createInitialPoint(key) returns point (x, y)
```

Figure 3. A Sample of Operations

There are other operations whose functionality requires no associated reverse operations. The operation *expand* (.) is an example of such an operation. It expands the key to appropriate length and consequently there is no need for an operation to reverse its

Copyright ⓒ 2017 SERSC Australia

functionality. Due to the importance of operations, we will discuss them in greater details in Subsection 2.2.

Each component has an identifier that identifies the component and its operations. Since we may have more than one operation in a component, the identifiers are multi-value variables. The way in which identifiers are represented is left to the implementations. For instance, we may give the *DistID* in the component *Distance* the values 50, 51, 52, ... , where the first digit 5 identifies the component Distance and the rest of the digits in each value identify the operations (*e.g.* 50, where 5 identifies the component Distance and 0 identifies the operation *rotateLeft* (.,.)).

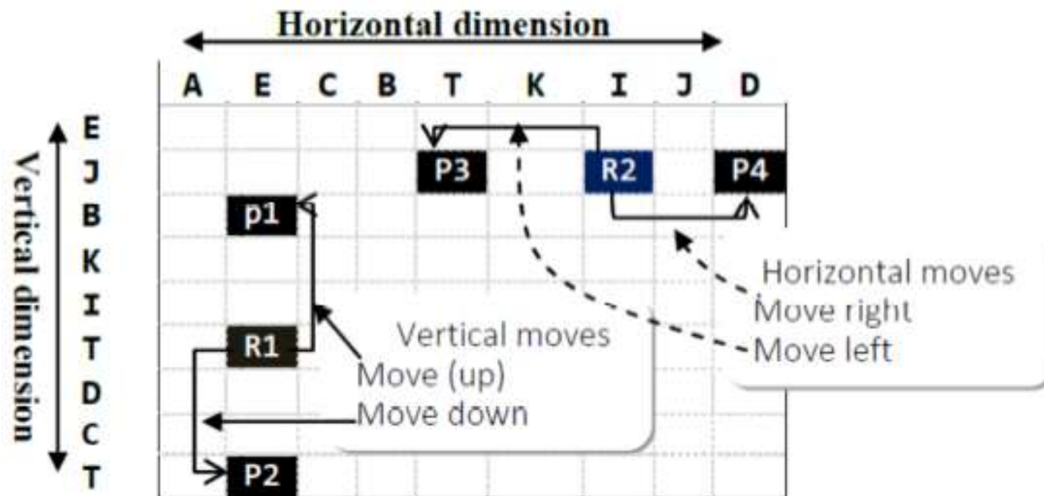

Figure 4. An Example of a Mesh

2.1. The Mesh

The mesh provides a mechanism for mapping symbols to directives and directives to symbols. Figure 4 shows an example of a mesh. The mesh is an $N \times N$ array with horizontal and vertical dimensions. Unicode symbols are listed in each dimension. The positions of the symbols in each dimension are indexed by non-negative integers 0, 1, ... For instance, the position of A in the horizontal dimension is 0 and the position of E in the same dimension is 1. Each cell in the mesh is a point (x, y), where x and y are the vertical and horizontal positions of a symbol respectively.

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

Since a move starts from some point and finishes at another, we introduce the move direction concept. We designate a direction of a move with respect to a point $P1$ by the flag "–" if the move is to a point with lower index, and by the flag "+" if the move is to a point with higher index. For instance, a move from point $R1$ to $P1$ is designated by "–" and the move from $R2$ to $P4$ is designated by "+". We may also introduce a dimension descriptor such as U to indicate one of the two dimensions of the mesh.

Given a distance of a move and its direction, we formalize our directives as follows. If we move from P to R by a distance x along the vertical dimension, we designate this by "$–xU$" or "$+xU$" depending on whether we move up or down from P respectively. Likewise, if we move from P to R by x along the horizontal dimension, we designate this by "$–x$" and "$+x$" depending on the direction of the move with respect to P. Observe that we omit the descriptor U here because its absence indicates the horizontal dimension.

 Copyright ⓒ 2017 SERSC Australia

After introducing the mesh and directives, we show in a way to be made precise later how to use the mesh to map symbols to directives and vice versa. The mesh maps a symbol to a directive by beginning from a starting point and moving along one of the dimensions to the position of this symbol. We call this dimension a *mapping dimension*. The distance of the move and its direction with respect to the starting point is compiled as a directive for that symbol. Mapping a directive to a symbol is performed by moving from a starting point along a specific dimension the number of positions equal to the distance and according to the specified direction. We look up the corresponding symbol from the dimension of the move. For instance, if the dimension of the move is vertical, we determine the symbol from the vertical dimension.

2.2. Operations

The operations support the dynamic behavior of our algorithm. Figure 3 shows three types of operations: masking operations for manipulating *Distance* and *Direction* instances, ordering operations for organizing Mesh symbols, and mapping operations for determining a dimension of a move within a mesh.

Masking Operations Masking operations mask a directive's distance and direction. They generally hide the actual directives and produce a false image of them.

Figure 3 provides examples of masking operations. Distance masking operations such as *rotateLeft* (.,.) and *shiftForward* (.,.) mask a directive's distance by applying key-based behavior. Implementers define this key-based behavior. For instance, they may define a behavior of the operation *rotateLeft* (.,.) to left rotate the value of distance *n* positions, where *n* depends on the value of the current key's symbol, and define a behavior of the operation *shiftForward* (.,.) to simply add the value of the current key's symbol to a distance. Direction masking operations such as *maskDirectionWRTPoint*(.) mask a directive's direction with respect to a point. These operations depend on the key to hide the actual direction of a move. Implementers may define the behavior of a direction masking operation to change the direction only if the value of the current key's symbol exceeds a pre-specified threshold. Table 1 provides examples of masking a distance of a move.

Table 1. Examples of Applications of Masking Operations

Program: +12U–31+29–3U+651U	**Key: 1234**
Forward shifting (add key digits)	+13U–33+32–7U+652U
Rotate left by 1 position	+21U–13+92–3U+516U

Using these operations, our algorithm can dynamically change the encryption by changing the behavior of masking operations. This change to behavior can be done by changing the behavior of the masking operations or by using different masking operations for different plain texts. For instance, modifying the behavior of *rotateLeft* (.,.) to left rotate the distance value only if the value of the current key's symbol is prime makes the masking output different from that in Table 1. In addition, as Table 1 shows, using different masking operation, *shiftForward*(.), yields a different encryption.

Ordering Operations Ordering operations impose organization (repositioning) of the symbols of each dimension of the mesh. Figure 3 shows an ordering operation, *order*(*dimension*, *key*). Implementers can define any key-based behavior for this operation. For instance, they can define the operation to reorganize the symbols of the mesh by moving ahead a symbol from its current position a number of positions equal to the value of the current key's symbol. As another example, they can define the behavior of this operation to generate controlled random numbers between 0 and the maximum number of symbols using a key as a seed, placing each symbol in the index specified by the value of the corresponding random number.

Copyright ⓒ 2017 SERSC Australia

Mapping any particular symbol of plain text depends on the organization of the symbols in a mesh. In fact, when we map plain text symbol to the mesh, the distance of the move from current position to the position of a symbol to be mapped depends on the position of that symbol in the mesh. Generally speaking, if the organization of mesh's symbols changes, the mapping does likewise; yielding a different sequence of directives for the same plain text.

Mapping Operations The mapping operations define how to use the mesh for generating directives. As discussed in previous sections, a directive has a distance along one of the two dimensions and a direction. Mapping operations select one of the dimensions of the mesh (called a *mapping dimension*) and next calculate a distance of a move to position of that symbol in the mesh. Figure 3 shows examples of mapping operations. The operations *selectDimensionKeyBased*(.) selects one of the dimension of the mesh according to a criterion that depends on a key. The operation *selectDimensionDistanceBased*(.) selects one of the dimensions of the mesh according to a criterion that depends on the distance. For instance, implementers can define a behavior of the operation *selectDimensionKeyBased*(.) to select the vertical dimension if the value of the current key symbol is odd and selects the horizontal dimension otherwise. The operation *distanceBetween*(.,.) computes a horizontal or vertical distance from the current position to the position of the symbol to be mapped.

Once the dimension is determined, the direction of the move with respect to a starting point isdetermined by the operation *directionWRTPoint*(.).

2.3. The Key

The key is a sequence of *n* Unicode symbols, where *n* is a non-negative integer. The key supports directive masking and a mesh's symbol ordering. In addition to these tasks, the key supports the creation of an initial starting point. The initial point is a secret point (x, y) with x and y are integers, and it provides a reference from which directive generation starts. Figure 3 defines the operation *createInitialPoint*(*key*)to create an initial starting point using a key. Implementers can define any behavior they want for this operation. Examples include selecting *n* digits from the key for x and *m* digits for y, create x as the greatest divisor of the first *k* digits and y as the first *m* odd digits, or apply rather complex mathematical functions to the key. Table 2 gives examples.

Table 2. Examples of Initial Starting Points

An example of a key: 9287654329117398	
Initial reference point	(928, 98), the first 3 digits and last 2 digits of the key.
Initial reference point	(96, 975), the floor of the square root of the first 4 digits ($\lfloor\sqrt{9287}\rfloor$) and the first three odd digits of the key.

The initial point provides not only a starting point for directive generation, but also strengthens the security of the encryption. If we start from different locations (in or out of the mesh's index range), the resulting distances will be different. Hence, starting the directive generation from two different initial points results in two different sequences of directives and consequently two different encryptions for the same plain text.

We conclude this subsection by pointing out that the key length is often shorter than that of plain text. To handle this difference, our algorithm defines the operation *expand*(.) for expanding a key. Although the behavior of this operation is left to implementers, we recommend the use of genetic algorithm techniques such as mutation and crossover operations [15].

The mutation enables the creation of a new version of a key by modifying some randomly designated symbols of the key. These candidate symbols can then be modified in many ways. One way is to complement the candidate symbols. For digits, take the 9*th*

 Copyright ⓒ 2017 SERSC Australia

or 10*th* complement. For alphabetic symbols, take the complement within the range of the alphabetic symbols. For instance, for capital letters we can take the Yth or Z^{th} complement.[2]

The crossover operation scrambles key symbols, yielding a new version of the key. The crossover typically selects random cut points within the key and swaps the blocks between these cut points. Table 3 shows examples of expanding a key using mutation and crossover operations.

Table 3. Examples of Creating New Versions of the Key Using Mutation and Crossover Operations. Note: The Candidate Symbols for Mutation Are Boldfaced and The Random Cut Points for Crossover Are Designated by "|".)

Original key	New key	The operation
98**7**65439**0**987**1**2**2**3	98**2**65439**9**987**8**2**7**3	Mutation
Secui**r**ty1**9**8A**dg**	**G**ecui**h**ty1**1**8A**vs**	Mutation
As8*j*976retreem*j*lk90	976retreemlk90As8	Crossover

3. System Overview

This section briefly describes the general architecture of the system (Figure 5). We focus on the general behavior of each component. Further discussion about the functionality of each component is deferred to following sections.

The *program generator* encrypts symbols of plain text in terms of directives. The generator takes a key and a mesh in addition to plain text as an input and maps each symbol to the mesh using the key to produce a directive for the symbol. The output of the program generator is a sequence of directives that encrypts plain text.

The *masker* distorts (masks) the directives for further protection against attacks. The input to this component is a sequence of directives and a key. The masker then applies masking operations to mask the distance and the direction flag of directives, yielding a masked sequence of directives.

The *binary generator* transforms the masked directives into a stream of 0's and 1's. The binary transformation can be specified by appropriately representing the directives. Implementers can specify how to represent the direction and the distance of the move. This translation to binary serves dual objectives. First, it adds further protection to the program because the binary code does not follow any specific pattern and leaves no clue that could enable hackers to decrypt the text. Second, it reduces the storage requirements of the resulting program, making it more efficient for network transmission and local storage.

The *binary decoder* transforms the binary representation of the program to a sequence of directives.

The *DeMasker* uses the key and operations to unmask directives. The functionality of this component will be made precise in Section 4.

The *program interpreter* uses a key and a mesh to decrypt ciphered text. It uses the key to produce the initial starting point and organize symbols of the mesh. The program interpreter maps each directive to the mesh and produces the corresponding plain symbol.

[2] The 9th or 10th complement of a decimal x is defined as (c – x), where c is either 9 or 10. The Yth or Z^{th} complement is defined as (L – C) + A, where L is either Y or Z and C is any capital letter. For instance, the Yth complement of B is (Y (121) –B (98)) + A (97) = X (120).

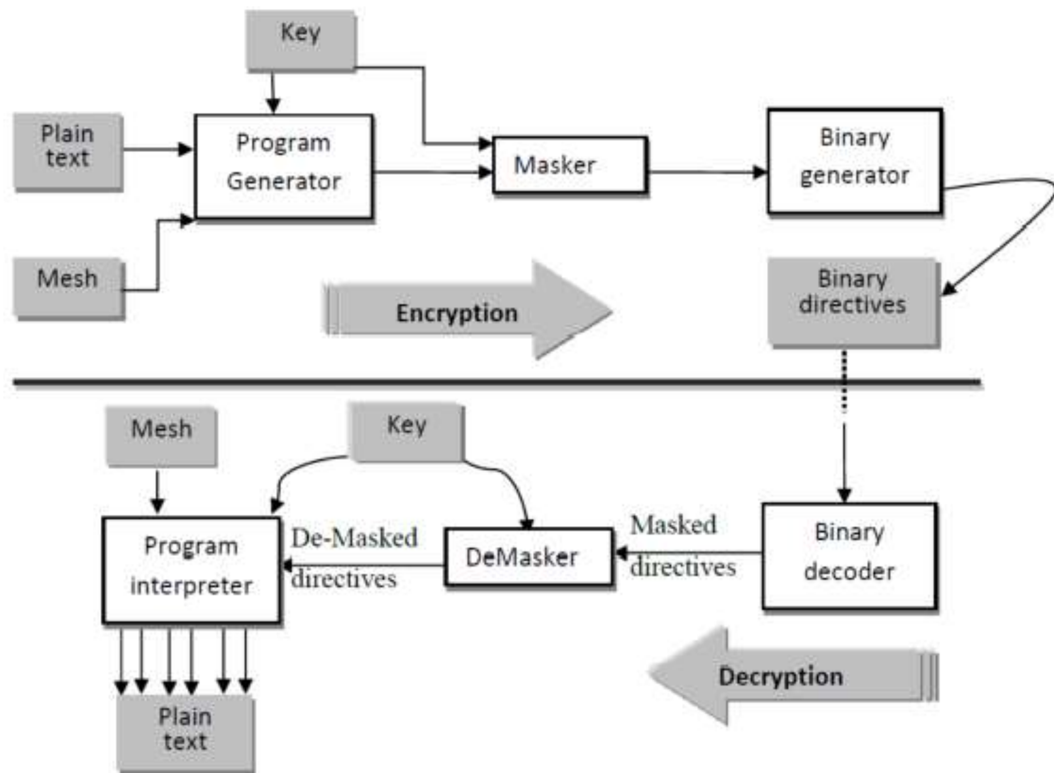

Figure 5. General Architecture of the System

4. The Encryption Process

The encryption process consists of three main steps. First, the program generator encrypts plain text in terms of a sequence of directives. Second, the masker masks further directives using masking operations. Finally, the binary encoder transforms the masked sequence of directives to a binary stream.

4.1. Program Generator

The program generator supports the main functionality of our algorithm. It generates a sequence of directives for plain text. To do its task, it uses a mesh and mapping operations.

The program generator reads plain text symbols one at a time, and maps each symbol to the mesh. The result of symbol mapping is (1) an integer that represents the distance of the move from the current position to the position of the symbol to be mapped, and (2) a direction flag that indicates the direction of the move with respect the current position, which is either the position of the last mapped symbol or the initial starting point. The distance can be measured along either the horizontal or vertical mapping dimension.

In order to strengthen the encryption, we define two states for selecting the mapping dimension: *distance-based state* and *key-based state*. When the program generator assumes a distance-based state, the operation *selectDimensionDistanceBased*(.) selects one of the two dimensions to be the mapping dimension according to a criterion that depends on the distance. For instance, the behavior of the operation may be defined to select the dimension that results in a shorter distance.

When the mapping dimension is selected, the location of the symbol to be mapped determines the direction of the move within the selected dimension. If the mapping dimension is horizontal, the direction is either to the left of the current position and we designate this by the direction flag "–", or to the right of the current position and we

 Copyright ⓒ 2017 SERSC Australia

designate this by the direction flag “+". Likewise, if the mapping dimension is vertical, the direction of the move is either up from the current position which we designate by the direction ag “–*U*" or otherwise is below the current position which we designate by the direction ag “+*U*”.

When the program generator assumes a key-based state, the operation *selectDimensionKeyBased*(*key*) selects one of the dimensions to be the mapping dimension based on the value of the current key symbol. With this state, we do not need to use the descriptor “*U*” since the mapping dimension can be inferred from the key.

To illustrate the program generator process, suppose we want to encrypt the plain text “To be or Not to be”. Suppose further that the initial starting point is (9, 8) and the key is “919981237659112348". Table 4 shows the programs for the plain text using the distance-based state and key-based state. Figure 6 shows the states of the mesh while generating directives for the first two symbols “T” and “o”.

Table 4. The Generated Program for the Plain Text

Plain Text: To be or Not to be.	
Distance-based state	–4+3–2U–1–1U+1U+1–5+0U+1U+5–3U+2U+2U–0+2U –1–1U–3
Key-based state	–6–1+5–3–0+3–1–6+6–7–5+2+2–1+7+2–1+2–7

The program generator reads the first symbol “T” from its input. Referring to Figure 6(a), the horizontal and vertical positions of the symbol “T” in the mesh are 4 and 3 respectively. The horizontal distance of the move from the initial starting point to the position of the symbol “T” is 4 while the vertical distance of the move is 6. If the program generator assumes the distance-based state, it selects the horizontal dimension as the mapping dimension since it yields a shorter distance. Now the distance of the move is 4. Because the location of the symbol “T” in the mesh is to the left of the initial starting point, the direction of the move is designated by the flag “–". As a result, the program generator encrypts “T” by the directive “–4". Now, the current point is (9, 4).

Using the same logic, the program generator generates directives for all the symbols. Figure 6(b) illustrates how to generate the directive for the letter “–o”.

The second row of Table 4 shows the output if the program generator assumes the key-based state. We use the same reference point and the same key. To illustrate the key-based state, let us suppose that the program generator reads the symbol “T” from its input and reads the leftmost digit “9” from the key. Selection of the mapping dimension depends on the value of the current key symbol and the behavior of the operation. Let us suppose that the operation *selectDimensionKeyBased*(*key*) has the behavior we described before, which we reproduce here: “if the value of the current-key digit is odd select the vertical dimension else select the horizontal dimension.” Because the value of the current key digit is 9, which is odd, this operation selects the vertical dimension to be the mapping dimension. The distance of the move from the initial starting point to the symbol to be mapped along the vertical dimension is 6. On other hand because the location of “T" within the mapping dimension is up that of the initial starting point, the program generator designates this as “–”. Therefore, the program generator encrypts “T” by the directive “–6”.

The program generator updates the starting point to (3, 8) and uses the new starting point to generate the directive for the symbol “o" using the same logic.

The program generator appends the *id* of the component and the operation that is used to determine the mapping dimension. This is important for the program interpreter to correctly interpret the program.

Copyright ⓒ 2017 SERSC Australia

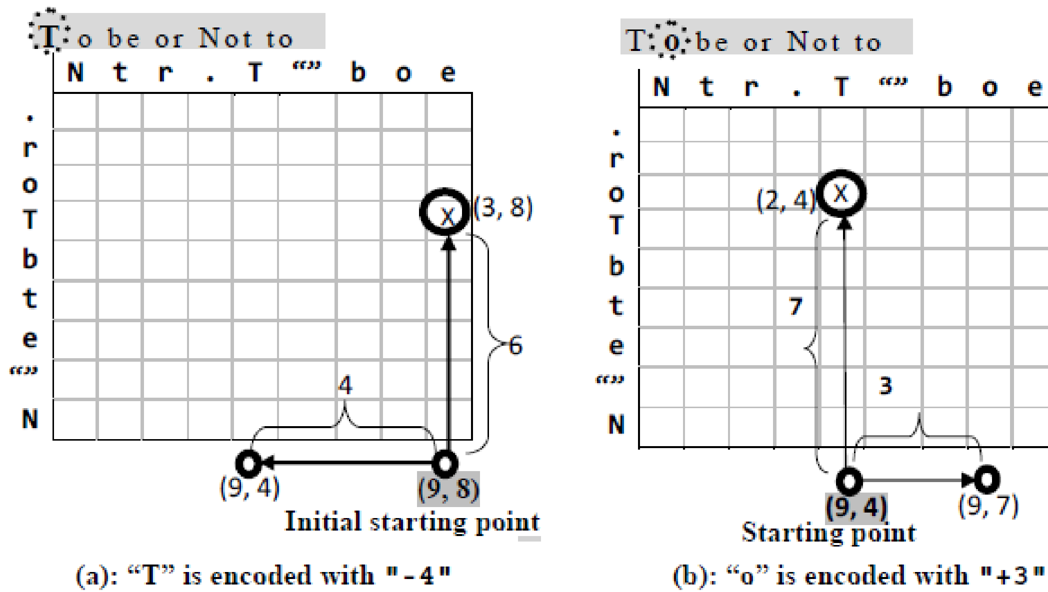

Figure 6. The Mesh Configuration during Mapping the Symbols “T” (left) and “o” (right)

4.2. The Masker

Masking consolidates the encryption. The masker receives a sequence of directives and a key as an input and applies masking operations to these directives. The output is further-masked directives.

To illustrate the masking process, suppose that the current behavior of the operation *rotateLeft* (.,.) is defined to left rotate the distance value's digits a number of positions equal to the value of the current key's symbol. Suppose also the current behavior of the operation *maskDirectionWRTPoint*(.) is defined to change the direction of a directive only if the value of the current key digit is odd. Consider the following directive “+145”. Based on the defined behavior of the two operations, if the current key digit is 1, the masker produces the masked directive “–451”. Observe that the masker rotated the distance by 1 to the left yielding the masked magnitude \451". Because the key digit is 1, which is odd, the operation for changing the direction changes the direction from “+” to “–”.

To further discuss the masking process, let us suppose that the behavior of the operation *rotateLeft* (.,.) is changed to left rotate the distance value's digits two positions if the value of the current key symbol is odd. Given this change, the operation produces a different masked directive *“*–514”. This means that even if we use the same key, the output of the masking depends on behavior of the masking operations.

Because the components may define any number of masking operations, the masker appends the *id* of the component and the operations that must be used to unmask the directives. In our example, the masker appends *ids* of the component *Direction* and *Distance* and the two operations *rotateRight*(.,.) and *unmaskDirectionWRTPoint*(.) as these operations reverse the behavior of the masking operations.

5. The Decryption Process

The decryption process is the reverse of the encryption process. It decrypts a sequence of directives to plain text. The input to the decryption process is a binary stream that contains a sequence of directives and control tags (the *ids* of objects and operations). The binary decoder transforms the binary stream to the standard representation.

 Copyright ⓒ 2017 SERSC Australia

The sequence of directives and control tags are then passed to the *DeMasker* to unmask the directives. To unmask directives, the *DeMasker* calls the operations referenced by the *ids* in the same order imposed by the order of the *ids* and passes the appropriate input to these operations. The *DeMasker* has no knowledge of the unmasking operations nor about their behavior. It only calls interface operations, passing to them the *ids* of the required masking operations and the appropriate input. In this case, implementers can freely change the behavior of the operations without affecting the encryption/decryption algorithm or recoding it.

The program interpreter receives the unmasked sequence of directives and *ids* along with the key as an input and interprets the program as plain text. Generally speaking, the program interpreter performs the following tasks. First, it re-creates the initial starting point from the key. Second, it uses the operation *order*(.,.) to organize the mesh symbols so that they match the organization of these symbols during the encryption. Third, the program interpreter calls the dimension mapping operation specified by the *id* to map each directive to the mesh and extract the corresponding symbol.

We use an example to illustrate the decryption process. Consider the program in the second row of Table 4, which was generated using the key-based state. Assuming that the operation that determines the mapping dimension of the move, *selectDimensionKeyBased*(.), has the behavior we defined before, the mesh state is as in Figure 6(a), and the key is "919981237659112348". Given this information, the program interpreter can decrypt a directive and recover the correct symbol as follows. The interpreter reads the first directive "−6" and the first symbol of the key "9". Since the key symbol is odd, the dimension mapping operation instructs the interpreter to use the vertical dimension. Next, because the directive is "−6", the interpreter moves vertically 6 indices lower than the starting point (9, 8) to reach the point (3, 8). Since the move is along the vertical dimension, the interpreter looks up the corresponding symbol from the vertical dimension of the mesh, which is the symbol "T" as indicated by Figure 6 (a).

Illustrative Example: The System in Action Consider the text "Encrypt me.". To keep the example simple and illustrative, we choose the simple initial starting point (9, 8) and the simple key "91763i5u745AE8". We assume that the operation *selectDimensionKeyBased*(*key*) selects the mapping dimension to be the vertical dimension if the value of the current key's symbol is odd and to be the horizontal dimension otherwise. We assume the algorithm uses the operation *shiftForward*(.,.) as the masking operation, which simply adds the value of the current key's symbol to the distance. We skip the binary generation and direction masking steps for simplicity. In addition, we present only the part of the mesh that contains the symbols that appear in the text.

Figure 7 shows the moves within the mesh when the program generator produces the program using the key-based state (solid arrows) and distance-based state (dashed arrows). The arrows depict the actual moves not the masked ones. (We added the indices to both dimensions to simplify the explanation.)

Initially, the program generator calls the operation *order*(.,.) to impose key-based ordering on the symbols of the mesh, yielding the symbol organization in Figure 7. Next, the program generator reads the first symbol "E" and the first key symbol "9". Based on the assumed behavior of the operation *selectDimensionKeyBased*(.), the mapping dimension is vertical. Therefore, the program generator moves from the initial starting point along the vertical dimension up to the vertical index of the "E". The distance of the move is 5. Therefore, the program generator outputs "−5" as a directive for the symbol "E". Continuing likewise, the program generator encrypts each symbol as shown by the solid arrows, yielding the program "−5−2+3−2+2−6−1+3+7+6−2". Since the program generator used the key-based state, it appends the *id* of the component and the operation, which is used to select the mapping dimension. Let us suppose that this *id* is 13. Therefore the output of the program generator is: "13−5−2+3−2+2−6−1+3+7+6−2".

Copyright ⓒ 2017 SERSC Australia

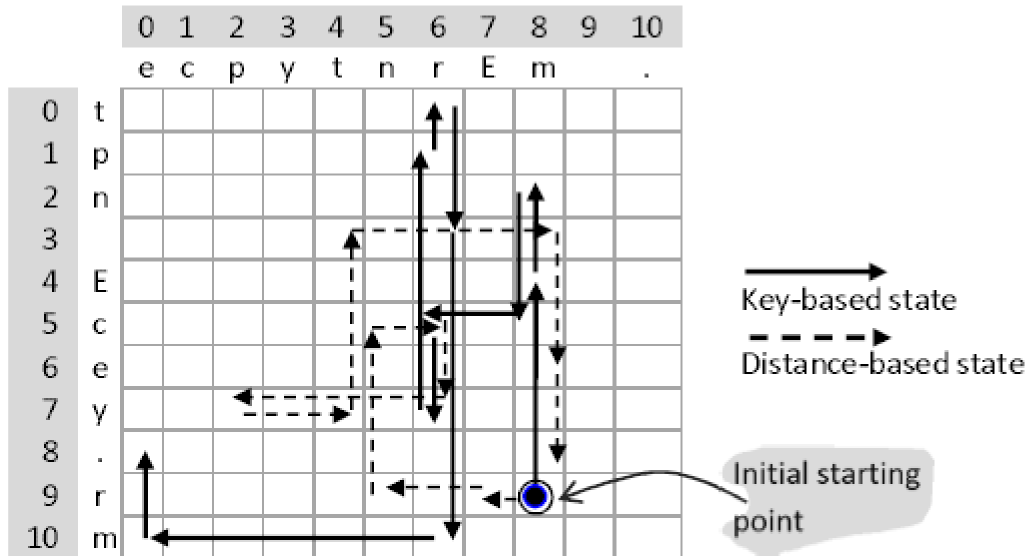

Figure 7. The Moves within the Mesh during the Creation of the Program. Solid Arrows Represent the Moves in Key-Based State and Dashed Arrows in Distance-Based State

The masker applies the operation *shiftForward*(.,.) to mask the distance part of every directive including the *id*. First, the masker reads the key symbol “9”. Therefore, it adds 9 to 13 to yield 22. Then it reads the second key symbol “1” and adds its value to the distance part of the directive “–5" to obtain “–6". Continuing in the same way, the masker outputs the masked program: “22–6–9+9–5+107–11–118+10+11+11–67”.

The masker finally appends a reference to the operation that unmasks the program. Referring to Figure 3, the operation *shiftBackward*(.,.) reverses the functionality of *shiftForward*(.,.). If we assume that the *id* is 63, the masker appends 63 to the program, yielding: “6322–6–9+9–5+107–11–118+10+11+11–67”.

The decryption process starts when the *DeMasker* receives the program as an input. The *DeMasker* strips the 63 part of the program, which references the operation *shiftBackward*(.,.). The *DeMasker* then calls this operation passing to it the id (22) and the distances of each directive and receives the unmasked *id* and distances. After unmasking, the program is “13–5–2+3–2+2–6–1+3+7+6–2.” The unmasked program is passed to the program interpreter.

The program interpreter first calls the ordering operation *order* (.,.) to organize the mesh's symbols to match their organization when the text was encrypted. Once this step is executed, the interpreter strips 13 from the program, which references the component *Generator* and the mapping operation *selectDimensionKeyBased*(.). The program is now “–5–2+3–2+2–6–1+3+7+6–2.” The program interpreter uses the mapping operation to obtain the mapping dimension for each directive and then looks up the symbols from the mapping dimension. In more details, the interpreter reads the key symbol “9”. Because the key symbol is odd, the mapping dimension is vertical. The currently read directive is “–5", which means that we have to move vertically up the starting point (due to the direction flag “–”) by 5 indices. Referring to Figure 7, the interpreter looks up the corresponding symbol from the vertical dimension (“E”). The next key symbol is 1 (odd); therefore the mapping dimension is vertical. The next read directive is “–2”, which means that we have to move up the index of the last decoded symbol “E" (that is from (4, 8) to (2, 8)). Referring to Figure 7, the interpreter looks up the corresponding symbol from the vertical dimension (“n”). Continuing in a similar way, the program interpreter decrypts the program, yielding the plain text “Encrypt me.”.

 Copyright ⓒ 2017 SERSC Australia

Figure 7 also shows the moves when the program generator assumes the distance-based state (dashed arrows). When the program reads the symbol “E” from the text, there are two possible moves: either to move vertically 5 indices up from the starting point or to move horizontally 1 index to the left of the starting point. Because the program generator uses the distance based state, it chooses to move horizontally and thus encrypts “E” by the directive “−1". Observe for the symbol “n” that the move is also horizontal as the vertical move has a larger distance, therefore the generator encrypts “n" by the directive “−2”. Proceeding in similar way yields the program “−1−2+4U+1−2U−4+2+4U+4−3U−2U".

Figure 7 indicates an interesting feature of the dynamic behavior of our system. The generated program differs as the behavior of the program generator changes. Even if the program generator uses the same key and the same mesh, it generates different programs for the same text. This difference is obvious by looking at the patterns of the moves (solid and dashed arrows).

6. Parallelizing the Algorithm

As discussed in Sections 4 and 5, the program generator and the program interpreter are the main components of the algorithm. All of the other components provide supporting functionality to them. Therefore, parallelizing the algorithm is achieved by parallelizing these two components. Figure 8 suggests the parallelized version of the algorithm.

As Figure 8 shows, the cipher divides the plain text into k chunks, preferably of the same size. It then spawns k threads of the program generator and assigns a chunk to each thread. Once all the threads finish encrypting the assigned subtexts, the results are combined based on the order at which the threads started.

The decryption process follows the same pattern of computations. Since each symbol is encrypted as a directive, it is possible to divide the set of directives also into k chunks (subprograms). These chunks are then assigned to k threads of the program interpreter. Once each thread decrypted the subprogram, the subtexts are combined into plain text.

7. Performance Analysis

We analyze our encryption algorithm in this section. This includes a study of the encryption performance (Subsection 7.1) followed by an analysis of the time complexity (Subsection 7.2).

7.1. Security Analysis

As presented, our approach uses the key as one of the tools to protect the security of the data. Unlike others, however, our encryption algorithm provides operations (or dynamic behavior) that work synergistically with the key to strengthen the encryption. These include operations for ordering symbols on the mesh dimensions, selecting a mapping dimension, masking directives, expanding a key, and creating a starting point from which the program generation begins.

Given this, knowing the key alone is far from sufficient to decrypt ciphertext, since the dynamic behavior adds more security guards to the ciphertext. First, the symbol ordering operation reorganizes the symbols on the two dimensions of the mesh. To reproduce the correct symbol organization, attackers must know the internal logic of this operation and how it uses the key to do the ordering. Random guessing is not possible given the tremendous number of possibilities. As a matter of fact, if we have n symbols in each dimension, the number of possible different organizations of these symbols on both mesh's dimensions is $n! * n!$ (n factorial or $n(n\text{-}1)(n\text{-}2)...3.2.1$). For instance, the number of different symbols in a moderate size of plain text may exceed 30 different symbols, and

Copyright ⓒ 2017 SERSC Australia

therefore, the number of possibilities for organizing the symbols on the two mesh dimensions is greater than 30! * 30! ≃7.1*10^{64}.

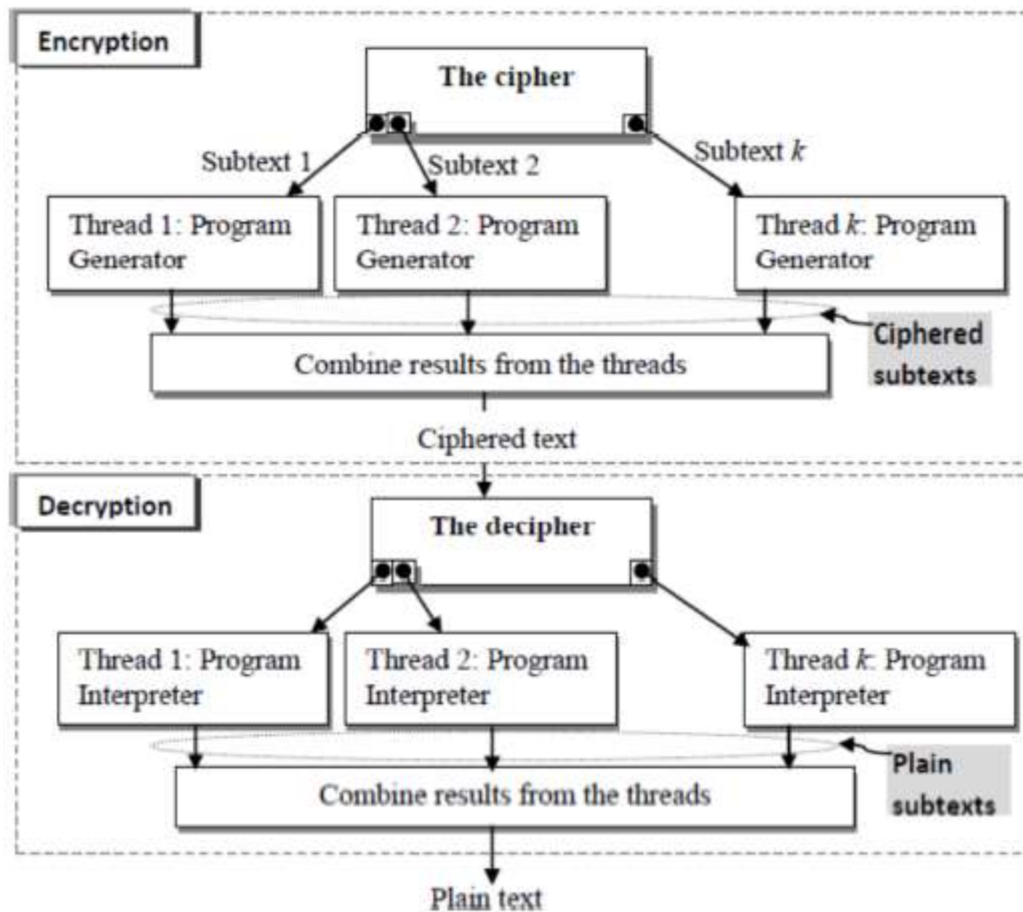

Figure 8. The Moves within the Mesh during the Creation of the Program. Solid Arrows Represent the Moves in Key-Based State and Dashed Arrows in Distance-Based State

Second, the operations that select the mapping dimension to be used for generating a directive (for a symbol) create another guard against attacks. Since the algorithm must choose one of the two mapping dimensions to produce each directive, the number of possible selections to produce directives for n plain text symbols is 2^n. This exponentially growing function becomes untraceable as n gets bigger.

Third, directive masking operations add further security guards to the ciphertext because they falsify the two parts of the directives, namely the distance and direction. As previously presented, masking operations change the value of the distance, yielding a new (false) value for the distance. They also mask the direction of a move within some mapping dimension.

To empirically measure the effectiveness of ordering and masking operations, we conducted several experiments on 5 different texts collected from Wikipedia with different sizes. To simplify the presentation we term the 5 texts as Text1 (31 symbols), Text2 (632 symbols), Text3 (1936 symbols), Text4 (3906 symbols) and Text5 (5000 symbols).

To measure the effectiveness of the ordering operation in strengthening the encryption, we turned off the masking operations and unified the starting point in all the experiments. We also used the same key in all the experiments, but expand it when the length of the key is less than the that of the text. We applied our algorithm 6 times for each plain text

 Copyright ⓒ 2017 SERSC Australia

(Text$_i$, i =1,2,..., 5). In each time, the ordering operation randomly reordered the symbols on the mesh's two dimensions. (In all of the experiments, and for the sake of simplicity, we only used the symbols of the plain text to appear in each of the mesh's dimensions. In general settings, however, we would use a larger set of symbols, not only those that appear in plain text.) We then grouped the resulting 6 ciphertexts in groups of two, yielding 15 different groups. For each group, we compared the distance part of the first ciphertext's directives with their corresponding directives in the second ciphertext and counted the directives for which the distances are different. The percentage of the number of the directives that differ is computed by dividing that number by the total number of directives. We did the same for direction part of the directives. Finally, we computed the average percentage among all the 15 groups in addition to other statistics. Table 5 shows the average difference percentages for each Text$_i$ and the maximum and minimum differences.

Table 5. The Average Percentage Difference between Ciphertexts as a Result of Applying Ordering Operation

	Difference in	Average percentage	Max percentage	Min percentage
Text$_1$ (31 symbols)	Direction	70.9%	92.3%	58.6%
	Distance	90.3%	100%	83.1%
Text$_2$ (632 symbols)	Direction	77.2%	96.0%	61.4%
	Distance	96.5%	100%	83.7%
Text$_3$ (1936 symbols)	Direction	77.6%	93.7%	62.2%
	Distance	97.1%	100%	83.8%
Text$_4$ (3906 symbols)	Direction	79.0%	93.1%	63.3%
	Distance	98.2%	100%	84.1%
Text$_5$ (5000 symbols)	Direction	81.7%	94.9%	63.7%
	Distance	98.8%	100%	84.5%
Overall	Direction	77.3%	94%	61.8%
	Distance	96.2%	100%	83.8%

As these statistics show, any change to the order of the symbols of the mesh causes a big change to the output regardless of the text size. This essentially means that ordering operation dissipated the statistical structure of plain text into a statistical structure involving long combination of symbols in the ciphertext. On the other hand, because the key is expanded as the algorithm produces more directives, the relation between the key and ciphertext becomes more complicated and involved one. In similar terminology as in block ciphers (*e.g.* Advanced Encryption Standard: AES), our algorithm has high diffusion and confusion, the two important properties for powerful ciphers [10][11].

To measure the impact of masking operations on the ciphertext, we implemented one operation for masking a directive's distance and one for masking a directive's direction. The distance masking operation has a simple behavior: it just adds a randomly generated number to the value of a directive's distance only when the random number is odd. The direction masking operation has also a simple behavior: it masks the direction only when the randomly generated number is even. (It should be clear that there exist a large number of ways to define the behavior of masking operations. We choose this behavior for our experimental masking operations just for showing the effectiveness of the masking.)

We randomly select one ciphertext for each of the 5 plain texts. We generated two random numbers: one to be used by the distance masking operation and the other to be used by the direction masking operation for each directive in a ciphertext. We next compared the directives of the original ciphertext with their corresponding directives of masked one and counted the directives that differ in either distance or direction or both.

Copyright ⓒ 2017 SERSC Australia

Finally, we computed percentage of the difference between directives of each original ciphertext and their corresponding directives in the masked one. Table 6 shows the difference percentages.

The distance masking operation falsified on average 59.4% of the directives and the direction masking operation falsified 58.1% as the numbers (in Table 6) show. That is, more than half of the directives of the ciphertext had been falsified. This masking, therefor, further distances ciphertext from the plain text and largely cuts its relationships to the plain text, raising the level of the security of the encryption and making it extremely hard for attackers to identify patterns that help them decrypt.

Table 6. The Difference Percentage between Directives of the Ciphertexts and Their Corresponding Directives after Applying the Masking Operations

	Difference Percentage	
Ciphertext	Distance	Direction
1	56.8%	52.1%
2	57.3%	56.0%
3	60.6%	59.1%
4	60.2%	61.4%
5	61.9%	61.7%
Average	59.4%	58.1%

To this end, this analysis shows that the dynamic behavior (ordering operation, mapping dimension selection operations, and masking operations) of our algorithm is effective in securing the encryption. As we discussed, this dynamic behavior causes high diffusion and confusion and highly melts relations to plain text.

7.2. Algorithmic Complexity

This subsection provides analytical study for the complexity of our approach. Our algorithm performs the following actions.

1. Mesh lookup: since the mesh is conceptually a two-way array, the system can directly access any symbol from either dimension using the index of that symbol. This simply requires one operation; that is the time complexity of this operation is of order O(1) in big-o notations.
2. Scanning the input: because the input is a string of length n characters, the system requires n accesses to fully scan it. Assuming that the system requires one operation to access one character in the string, it is clear that time complexity is linear in n. In big-o notation, the time complexity is of order $O(n)$.
3. Swapping: swapping the order of characters in the mesh or swapping the key blocks (crossover operation) are also of $O(k)$ time complexity, where k is the number of swaps.
4. Computations: computing the distance of a move, determining the direction of a move, and doing the masking are all simple operations that require only constant time.

As a result, the time complexity of the algorithm is linear.

Further, the space requirement is fixed. The system uses the following data: (1) the input string and (2) the mesh. The input string consists of n characters and therefore, this string needs a space of order $O(n)$. For the mesh, we do not store any value in the mesh. We need to store only the characters of the horizontal and vertical dimensions. Good implementation will then use two **one-way arrays**: one for the horizontal dimension and one for the vertical one. Therefore, the required space is the amount of memory required

 Copyright ⓒ 2017 SERSC Australia

for storing these two one-way arrays, which is clearly linear in the number of characters. Thus, in both cases the space requirements are the linear in the number of characters.

Table 7. The Execution Time in Milliseconds (Ms) For the Sequential and Parallel Implementation. The Input Size Is In Characters

	Sequential	Parallel	
Input size		**3 Threads**	**5 Threads**
1000	30 ms	30 ms	28 ms
5000	62 ms	54 ms	49 ms
10000	102 ms	92 ms	86 ms

We have also implemented the system using Java programming language and run it on a computer with processor Intel core i5 3470 (3.2 GHz). Table 7 shows the execution time for the sequential and the parallel implementations of our algorithm and for different input sizes. As the figures in Table 7 show, the parallel implementation outperforms the sequential implementation for large inputs (5000 and 10000). The performance of the parallel implementation, however, cannot be distinguished from that of the sequential implementation for small input (1000). For small inputs the extra overhead for collecting the outputs of the threads and putting them together negates the gains of the parallelism.

8. Conclusions and Directions for Future Work

The successive operations of this encryption algorithm that constitute the sequence generation based on the key and some initial plain text, in addition to the mesh itself, is a dynamic process by itself. Those dynamics make the algorithm more elusive and difficult to analyze. During the generation of the encrypted sequence, the next state of the mesh is a function of the current state and some key values. The mesh matrix itself with its existing sequence of symbols (vertical and horizontal) provides a rich environment for the generator to create the unrelated sequences of symbols. Usually static mappings are much easier to crack down, especially with abundant samples of encrypted/decrypted pairs of symbols. Look up tables, shuffling, and masking could be tracked with brute force and plenty of luck. Of course, in encryption, we cannot afford to have even minor chances of being uncovered or beaten. The fact that the algorithm is parallel in nature and that the whole process of encryption or decryption could be divided into several parallel batches that could be executed on different threads gives a great advantage for this technique. With parallel array-processor based hardware we can maximize the efficiency of that algorithm. Hardware implementation for it, as a possible future work, will maximize its speed and accuracy. The algorithm complexity being only O (n) means much less execution time and much less instructions per second are needed. We are talking about a "green" algorithm that will minimize heat dissipation and lost energy. In general, the algorithm is simpler than standard algorithms such as AES. With long keys, it can be as efficient as them also. As shown in Table 5, the effect of ordering on the cipher text is a direct proof about the capabilities of this algorithm to maintain high randomness and high muddle within the encrypted symbols.

We have two main directions for future work. First, the output of the algorithm (the ciphered text) is larger than the original. We are exploring different compression techniques to shrink the output size, making it more efficient for network transmission. Second, and perhaps more important, we are exploring ways to enable the decryption process to recognize the operations that must be used to interpret ciphered text without having to explicitly refer to these operations in the ciphered text itself.

Copyright ⓒ 2017 SERSC Australia

Acknowledgments

We highly appreciate David W. Embley and Kent Seamons from the Department of Computer Science, Brigham Young University, USA, for their great feedback.

References

[1] S. Bhati, A. Bhati and S. K. Sharma, "A New Approach towards Encryption Schemes: Byte Rotation Encryption Algorithm", Proceedings of the World Congress on Engineering and Computer Science, vol. 2, **(2012)**, pp.979-982.
[2] S. Natarajan, M. Ganesan and K. Ganesan, "A Novel Approach for Data Security Enhancement Using Multi Level Encryption Scheme", International Journal of Computer Science and Information Technologies, vol. 2, **(2011)**, pp. 469-473.
[3] G. Manikandan, N. Sairam and M. Kamarasan, "A New Approach for Improving Data Security using Iterative Blowfish Algorithm", Research Journal of Applied Sciences, Engineering And Technology, vol. 4, **(2012)**, pp.603-607.
[4] U. Devi and R. S. D. Wahida Banu, "Secure Multilevel Cryptography Using Graceful Codes", International Journal of Information and Electronics Engineering, vol. 2, **(2012)**, pp.840-843.
[5] S. Singh, S. Maakar and S. Kumar, "Enhancing the Security of DES Algorithm Using Transposition Cryptography Techniques", International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, **(2013)**, pp.464-471.
[6] H. P. Singh, S. Verma and S. Mishra, "Secure-International Data Encryption Algorithm", Secure-International Data Encryption Algorithm in International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, vol. 2, **(2013)**, pp.780-792.
[7] B. Saini, "Implementation of AES using S-BOX rotation", International journal of advanced research in computer science and software engineering, vol. 4, **(2014)**, pp. 1322-1326.
[8] S. Kashyap and Er. Richa, "An Enhanced Symmetric-Key Block Cipher Algorithm to Manage Network Security", International Journal of Computer Science, vol. 3, **(2015)**, pp.58-65.
[9] H. P. Singh, S. Verma and S. Mishra, "An Enhancement in International Data Encryption Algorithm for Increasing Security", International Journal of Application or Innovation in Engineering and Management (IJAIEM), vol. 3, **(2014)**, pp.64-70.
[10] C. E. Shannon, "Communication Theory of Secrecy Systems", Bell System Technical Journal, vol. 28, pp.656-715, **(1949)**.
[11] C. E. Shannon, "A Mathematical Theory of Cryptography", Bell System Technical Journal, vol. 27, **(1945)**, pp.379-423, 623-656.
[12] W. Stalling, "Cryptography and Network Security: Principles and Practices", (7th edition) Pearson, **(2016)**.
[13] J. Erickson, "Hacking: The Art of Exploitation", (2nd edition), No Starch Press, **(2008)**.
[14] D. W. Embley, B. K. Kurtiz and S. N. Woodfield, "Object-Oriented Systems Analysis: A Model Driven Approach", Yourdon Press, Englewood Cliffs, New Jersey, **(1992)**.
[15] M. Mitchell, "An Introduction to Genetic Algorithms", MIT Press Cambridge, MA, **(1998)**.
[16] "Advanced Encryption Standard (AES)", FIPS, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf/, **(2001)**.
[17] "Data Encryption Standard (DES)", FIPS, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf/, **(1999)**.
[18] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Saffor and N. Zunic, "The Mars Encryption Algorithm", IBM, **(1999)**.
[19] R.L. Rivest, "The RC5 encryption algorithm", in: Workshop on Fast Software Encryption, **(1995)**, pp. 86-96.
[20] M. J. Al-Muhammed and R. Abuzitar, "K-Lookback Random-Based Text Encryption Technique", in press in Journal of King Saud University, Computer and Information Sciences, https://doi.org/10.1016/j.jksuci.2017.10.002, **(2017)**.

 Copyright ⓒ 2017 SERSC Australia