Key generator using neuronal network and symmetric text ciphering in the mobile system communication

Tianandrasana Romeo Rajaonarison[1], Paul Auguste Randriamitantsoa[2]

[1] *Doctoral School in Sciences and Technology of Engineering and Innovation, Research Laboratory Telecommunication, Automatic, Signal and Images, University of Antananarivo, BP 1500, Antananarivo101 – Madagascar*

[2] *Doctoral School in Sciences and Technology of Engineering and Innovation, Research Laboratory Telecommunication, Automatic, Signal and Images, University of Antananarivo, BP 1500, Antananarivo101 - Madagascar*

ABSTRACT

The mobile communication system uses ciphering techniques to protect the data. On symmetric encryption, the keys on the emitter and receiver are the same. The neural network could be used for generating the key.. In this article, we use text and it will be ciphered with symmetric encryption like : Serpent, Advanced Encryption Standard (AES). After simulation, the 128 bit key generated will be analyzed using similarities between them. The selection of type of cyphering text algorithm is important. The receiver could not decipher with the Serpent algorithm if the emitter ciphers it with AES and vise-versa.

Keyword : - *Serpent, AES, neuronal, ciphering, text*

1. INTRODUCTION

The mobile system communication uses secret key encryption. So, the encryption key can be calculated from the decryption key or vice versa. The fig 1 explains the method of symmetric key encryption

- In general, they are equal: K = K'. This is symmetric encryption.
- Generally, only one algorithm is used for encryption and decryption: D = E
- The secret key is secretly shared between sender A represented by Alice and receiver B represented by Bob.

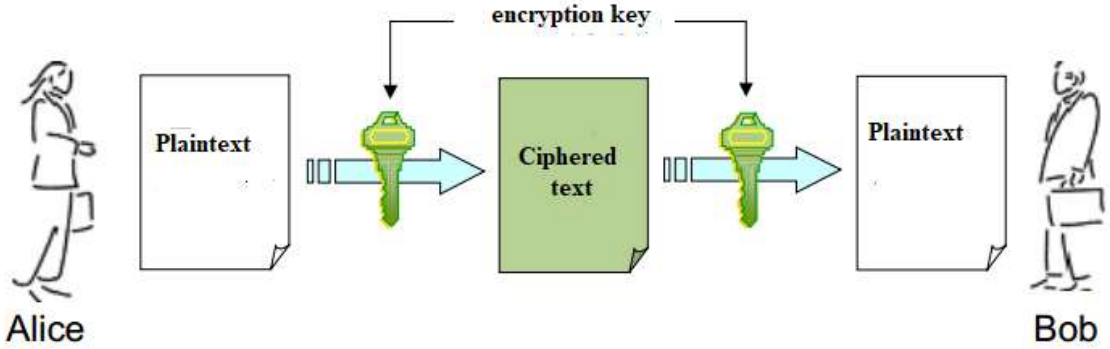

Fig -1: Symmetric key encryption

2. SECURING TEXT USING COMBINED NEURONAL CRYPTOGRAPHY

2.1 princip of ciphering

The neural key exchange algorithm is a synchronization application. Both partners A and B use a TPM machine (Tree Parity Machine) with the same structure. The parameters K (number of hidden layer units), L (the range of synaptic weight values made by the two machines A and B) and N (the input layer units for each hidden layer unit) are public .

Each TPM machine is initialized with randomly selected weights that must be kept secret. During the synchronization process, only the input vectors and the output vectors are transmitted on the public channel. Therefore, each user just knows the internal representation of their own TPM, which opens a new vision in the world of contemporary cryptography. Keeping information secret is essential for the security of the key exchange protocol. After the synchronization is complete, A and B use the modified weights as a common secret key which is thus used for AES encryption.

The main problem of attacker E is that it completely ignores the internal representation of A and the TPM of B. While the movement of the weights depends only on σi, it is thus difficult for an attack to correctly guess the state of the hidden units.

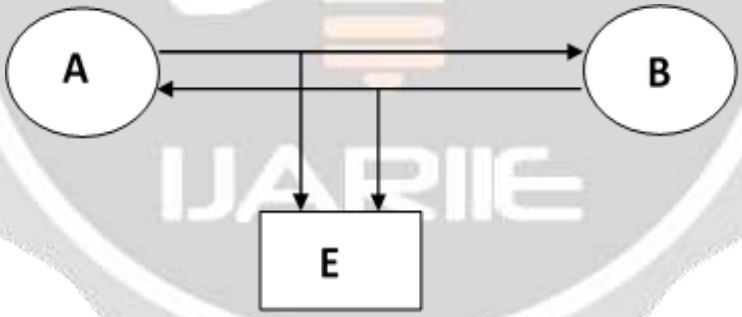

Fig -2: Attack of the secret key by E

2.2 Neural cryptography

The Diffie-Hellman key exchange protocol was introduced by Whitfield Diffie and Martin Hellman in 1976, to solve the problem of exchanging keys over an insecure channel in symmetric encryption. However, the Diffie-Hellman key exchange is vulnerable to a man-in-the-middle attack. In this attack, an adversary E intercepts Alice's public key and sends his own public key to Bob.

When Bob transmits his public key, Eve replaces it with her own key and sends it to Alice. Eve and Alice thus agree on a shared key and Eve and Bob agree on another shared key. After this exchange, Eve simply decrypts any

messages sent by Alice or Bob, then reads and, optionally, modifies them before re-encrypting with the appropriate key and transmitting them to the other party.

To remedy the problem posed above, we will build two neural networks, one for each user. Then they have to synchronize their networks, and the weights will be the secret keys of the TPM. The type of network chosen here is the forward-flow multilayer perceptron and the learning is in supervised mode. Here is a simple neural network developed by Rosenblatt in 1968

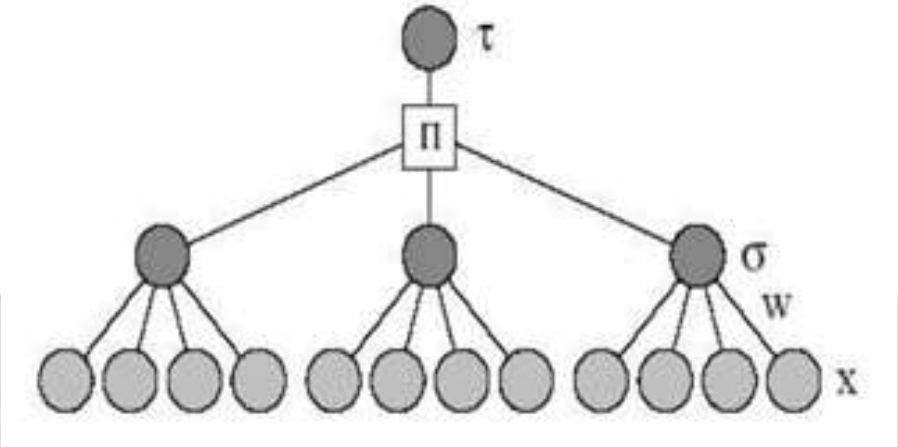

Fig -3: Gradient backpropagation perceptron mode

This model is a multi-layer forward-flow network (see Figure 3).

Supervised learning in this case consists of measuring the error between the inputs and the outputs and then propagating the error to the neurons of the hidden layers and those of the inputs.

It consists of an input vector X, a hidden layer Sigma σ, a weight coefficient W between the input vector and the hidden layer, and an activation procedure which counts the result value τ. Let's call such a neural network a neural machine. It can be described by three parameters: K: the number of hidden neurons, N: the number of input neurons connected to each hidden neuron, and L: the maximum value for a weight {-L,..+L}. Two partners have the same neural machines.

To count the output value, we use a simple method:

$$\tau = \prod_{i=1}^{K} SIGN[\sum_{j=1}^{N} w_{i,j} x_{i,j}] \quad (1)$$

A question arises, how can we update the weights? We update the weights when the output values of the neural machines are equal.

There are three different rules:

Table -1: learning rules

$w_{i,j}^{+} = g(w_{i,j} + x_{i,j}\tau\theta(\tau^{A}\tau^{B}))$	Hebb's learning rule
$w_{i,j}^{+} = g(w_{i,j} - x_{i,j}\tau\theta(\sigma_i\tau)\theta(\tau^{A}\tau^{B}))$	Anti- Hebb's learning rule
$w_{i,j}^{+} = g(w_{i,j} + x_{i,j}\theta(\sigma_i\tau)\theta(\tau^{A}\tau^{B}))$	Random-walk's learing rule

Here, θ is a special function. θ verified :

(a, b) = 0 if a <> b; others θ = 1.

The g function keep the weight in the gamma -L … + L. x is the input vector et w is the ponderation vector. After the machines are synchronized, their weights are equal: we can use them to build a shared key that will be impossible to hack due to the existence of chaotic synchronization.

2.3 Syncrhonization algorithm

.We will follow the following steps for neural key generation which is based on neural networks:

1. First determine the neural network parameters namely: k, N and L

2. Random initialization of neural network weights A and B.

3. Repeat 4 to 7 until synchronization occurs.

4. Calculated the inputs of the hidden layers.

5. The output bit is generated and exchanged between the two machines A and B.

6. If the output vectors of the two machines are equal i.e. τA = τB then the corresponding weights are modified using the learning rule of Hebb and Anti-Hebbian.

7. Once synchronization is complete, the synaptic weights are the same for both networks so these weights are used as the secret key.

This key is subsequently used to encrypt a text to be transmitted over an unsecured channel using the AES encryption algorithm with a key size of 128 bits, 192 bits and 256 bits.

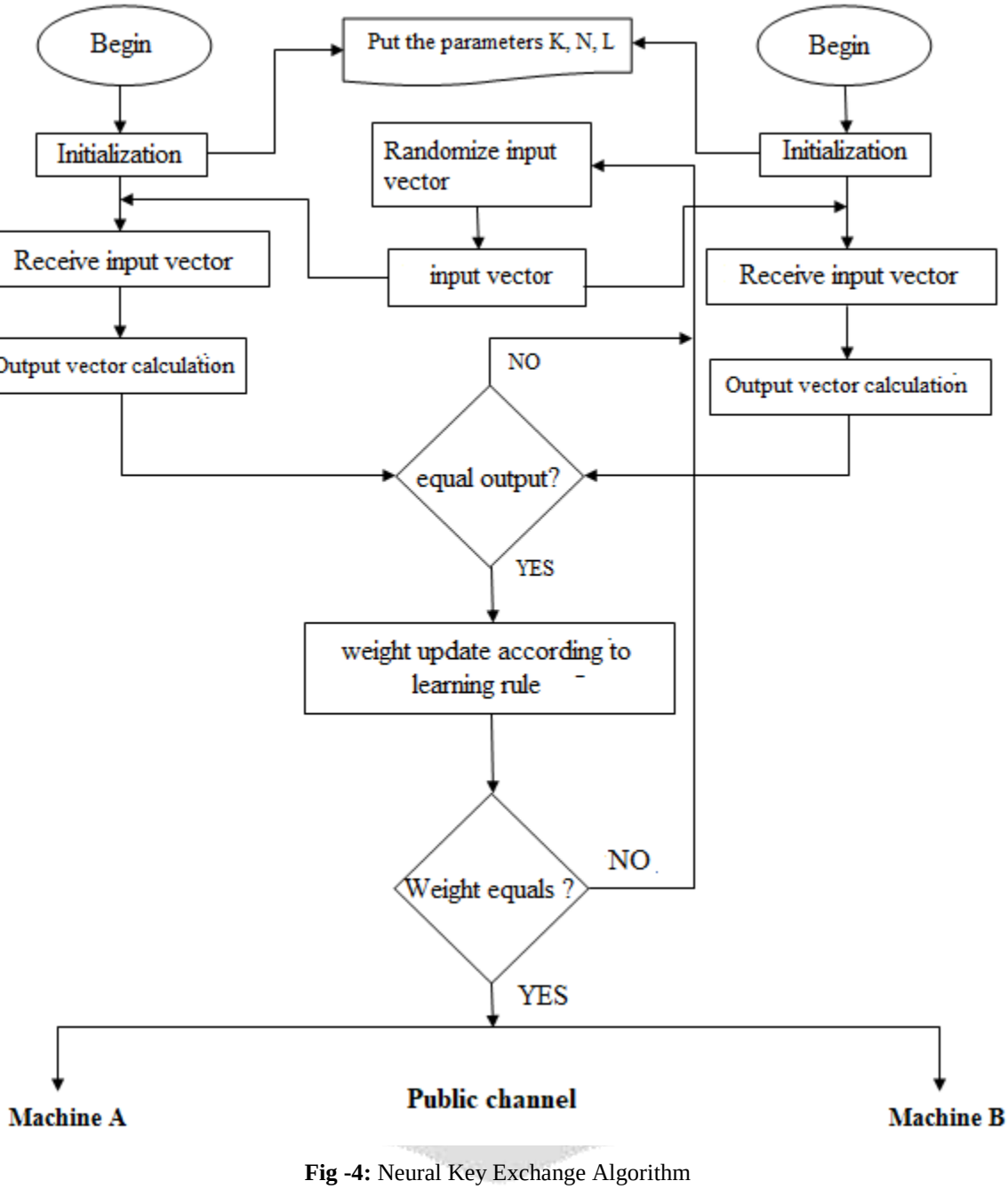

Fig -4: Neural Key Exchange Algorithm

2.4 Serpent algorithm

The Serpent key schedule consists of 3 main stages. In the first stage the key in initialized by adding padding if necessary. This is done in order to make short keys map to long keys of 256-bits, one "1" bit is appended to the end of the short key followed by "0" bits until the short key is mapped to a long key length.

In the next phase, the "prekeys" are derived using the previously initialized key. 32-bit key parts or XORed, the *FRAC* which is the fraction of the Golden ratio and the round index is XORed with the key parts, the result of the XOR operation is rotated to left by 11. The *FRAC* and round index were added to achieve an even distribution of the keys bits during the rounds.

Finally the "subkeys" are derived from the previously generated "prekeys". This results in a total of 33 128-bit "subkeys"

At the end the round key or "subkey" are placed in the "initial permutation IP" to place the key bits in the correct column.

```
#define FRAC 0x9e3779b9     //  fractional part of the golden ratio
#define ROTL(A, n) (A << n) | (A >> (32 - n))

uint32_t words[132]; // w
uint32_t subkey[33][4] // sk

/* key schedule: get prekeys */
void w(uint32_t *w) {
          for (short i = 8; i < 140; i++) {
                    w[i] = ROTL((w[i - 8] ^ w[i - 5] ^ w[i - 3] ^ w[i - 1] ^ FRAC ^ (i - 8)), 11);
          }
}

/* key schedule: get subkeys */
void k(uint32 t *w, uint32 t (*sk)[4]) {

          uint8_t i, p, j, s, k;

          for (i = 0; i < 33; i++) {
                    p = (32 + 3 - i) % 32;
                    for (k = 0; k < 32; k++) {
                              s = S[p % 8][((w[4 * i + 0] >> k) & 0x1) << 0 |
                                                  ((w[4 * i + 1] >> k) & 0x1) << 1 |
                                                  ((w[4 * i + 2] >> k) & 0x1) << 2 |
                                                  ((w[4 * i + 3] >> k) & 0x1) << 3 ];
                              for (j = 0; j < 4; j++) {
                                        sk[i][j] |= ((s >> j) & 0x1) << k;
                              }
                    }
          }
}
```

2.4.1 S-Boxes

The Serpent s-boxes are 4-bit permutations, and subject to the following properties:

- a 1-bit input difference will never lead to a 1-bit output difference, a differential characteristic has a probability of 1:4 or less
- linear characteristics have a probability between 1:2 and 1:4, linear relationship between input and output bits has a probability between 1:2 and 1:8.
- the nonlinear order of the output bits as function of the input bits is 3. However there have been output bits found which in function of the input bits have an order of only 2.

The Serpent s-boxes have been constructed based on the 32 rows of the DES s-boxes. These were transformed by swapping entries, resulting arrays with desired properties were stored as the Serpent s-boxes. This process was repeated until a total of 8 s-boxes were found.

2.4.2 Initial permutation (IP)

The initial permutation works on 128 bits at a time moving bits around.

```
for i in 0 .. 127
   swap( bit(i), bit((32 * i) % 127) )
```

2.4.3 Final permutation (FP)

The final permutation works on 128 bits at a time moving bits around.

```
for i in 0 .. 127
   swap( bit(i), bit((2 * i) % 127) )
```

2.4.3 Linear transformation (LT)

Consists of XOR, S-Box, bit shift left and bit rotate left operations. These operations are performed on 4 32-bit words.

```
for (short i = 0; i < 4; i++) {
    X[i] = S[i][B[i] ^ K[i]];
}
X[0] = ROTL(X[0], 13);
X[2] = ROTL(X[2], 3 );
X[1] = X[1] ^ X[0] ^ X[2];
X[3] = X[3] ^ X[2] ^ (X[0] << 3);
X[1] = ROTL(X[1], 1 );
X[3] = ROTL(X[3], 7 );
X[0] = X[0] ^ X[1] ^ X[3];
X[2] = X[2] ^ X[3] ^ (X[1] << 7);
X[0] = ROTL(X[0], 5 );
X[2] = ROTL(X[2], 22);
for (short i = 0; i < 4; i++) {
    B[i + 1] = X[i];
}
```

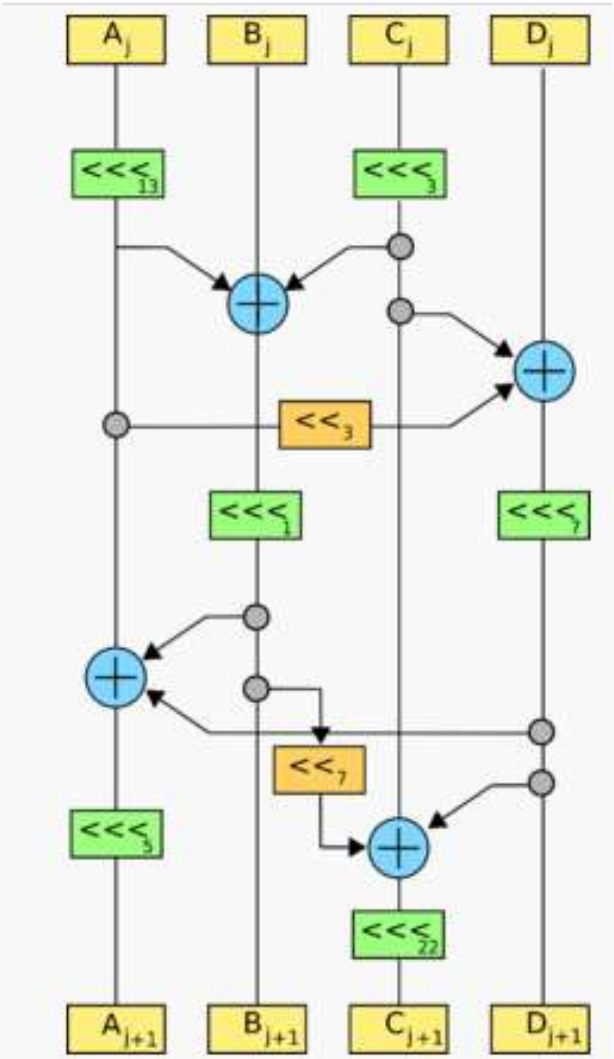

Fig -5: Serpent visualization

The fig 5 resume all process for having ciphering text with Serpent algorithm.

2.5 AES

The **Rijndael scheme is a block cipher** and divides the incoming plaintext into a **block with four rows and four columns**. A block has a **total of 16 bytes**, so each box contains one byte.

Each block goes through several rounds of four steps during **ciphering or deciphering**. Depending on the key length, there are ten rounds for AES-128, twelve rounds for AES-192 and a total of 14 rounds for AES-256.

The four steps are repeated in each round.

The symmetrical encryption procedure with AES in four steps

- Step 1 – **SubKeys**: Here, Rijndael uses an S-box. This indicates with which value the algorithm replaces which byte in the blocks. The S-Box is derived from the AES key.
- Step 2 – **ShiftRow**: Now Rijndael shifts the bytes in the blocks line by line by a certain number of columns to the left.
- Step 3 – **MixColumn**: At this point, the AES algorithm mixes the bytes using a mathematical procedure called a linear transformation.
- Step 4 – **AddRoundKey**: AES finally links the current round key with the values of the blocks.

Encryption with the Advanced Encryption Standard goes through these four steps as often as the key length dictates. The result is the so-called **ciphertext**, which no longer reveals anything of the content of the message or information to the naked eye. During **decryption**, Rijndael goes through all the steps over all the **rounds in reverse order**. This is how the **plaintext** is created from the ciphertext.

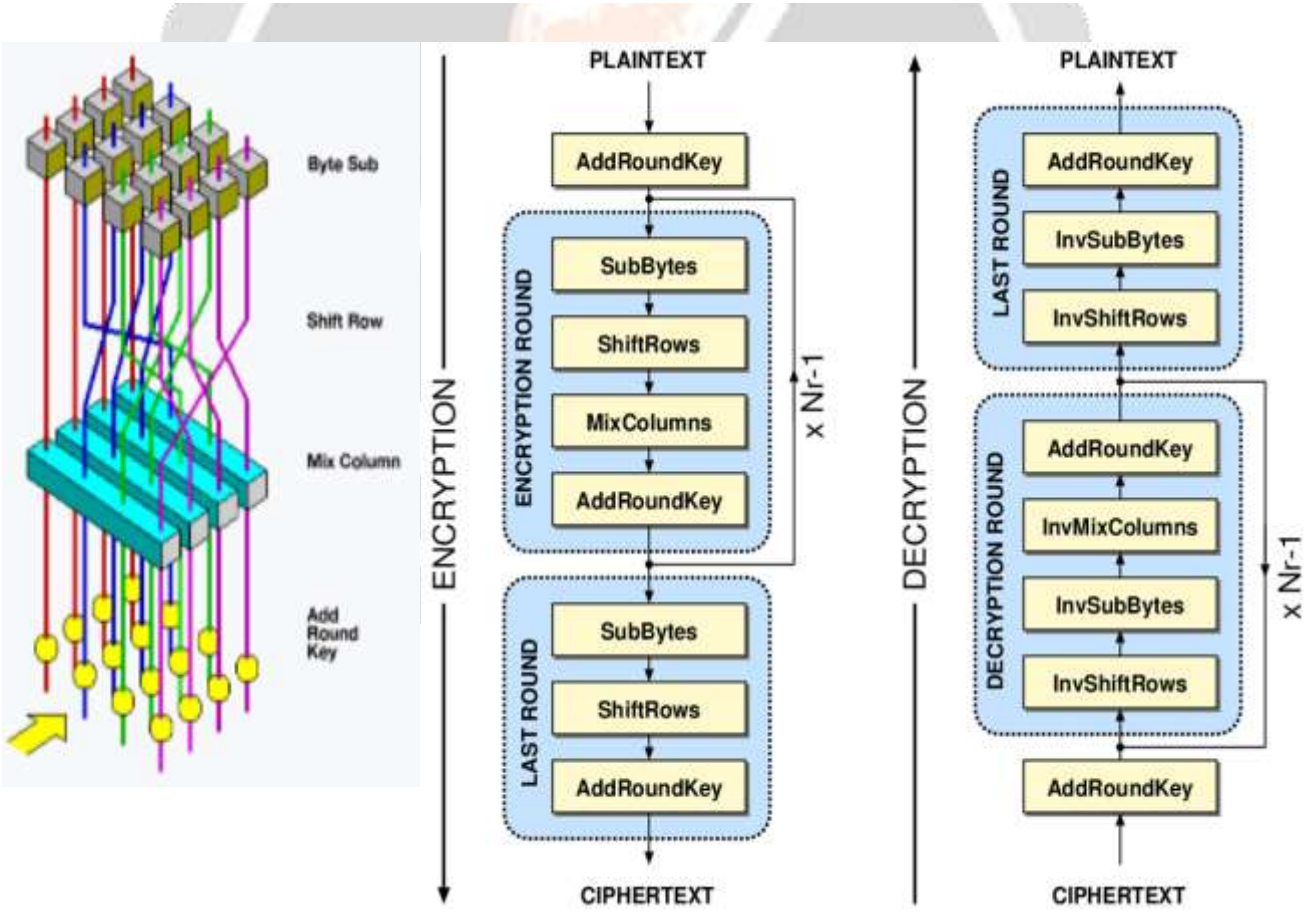

Fig -6: AES visualization

3. RESULTS

3.1 Interface of the software

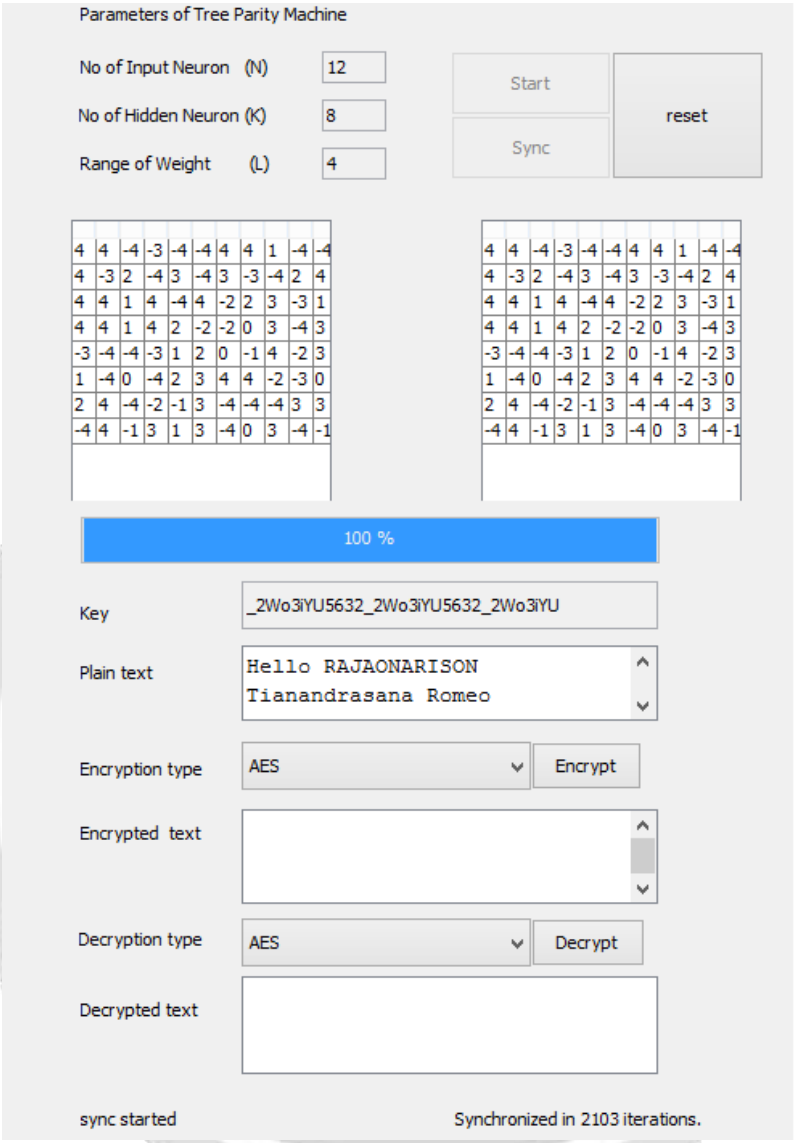

Fig -6: Software element

In this figure, the emitter clicks from reset -> start -> sync to synchronize to the generating key with 12bit. The parameters N, K and L should be complete. In this example, it is resp. 12, 8, 4. should

Before encrypting, the emitter should add the text to be sent in the input text labelled “Plain text”.

After clicking the button, our software gives the “Encrypted text” and it can be seen in figure 7.

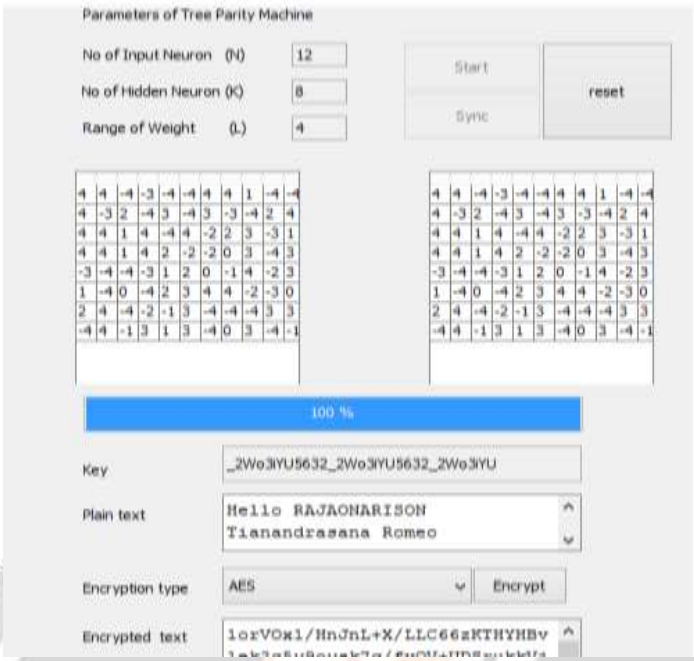

Fig -7: Encrypted text

3.2 Bad algorithm of decyphering

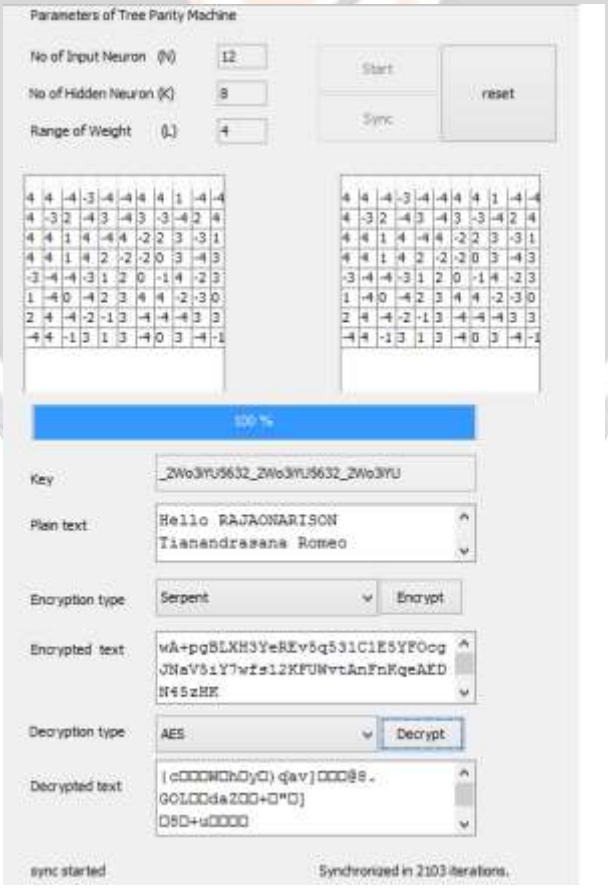

Fig -8: Bad algorithm of deciphering

Like in this case, the text is ciphered with AES. In addition, instead of deciphering the text with AES methods, we deciphered it with Serpent. The algorithm runs correctly but it doesn't give the plain text. So, with multiple algorithms, the type of algorithm of ciphering makes it more secure. The attacker must know the ciphering method before breaking the message.

3.3 Evaluation of similarity of key

As the key is 128bits, it will be represented by 32 hexers. The list of keys generated by number of iterations is represented by figure 9.

```
N=8, K=12, L=4
OZY5J867XS52OZY5J867XS52OZY5J867        959
3RZ7Z361VT583RZ7Z361VT583RZ7Z361        1677
899UY6_b2cJU899UY6_b2cJU899UY6_b        1689
ddX1WU_258Y0ddX1WU_258Y0ddX1WU_2        554
62KS2T131dcZ62KS2T131dcZ62KS2T13        2061
0m5SY85V747_0m5SY85V747_0m5SY85V        1557
7bd9Y1Z0OXYe7bd9Y1Z0OXYe7bd9Y1Z0        1154
3Y0RdX8W6WUS3Y0RdX8W6WUS3Y0RdX8W        3823
7RfUTb2Q7jT27RfUTb2Q7jT27RfUTb2Q        1485
VP390Z2h7QXVVP390Z2h7QXVVP390Z2h        751
ZU0X73gS1Q4gZU0X73gS1Q4gZU0X73gS        1848
0b2OVVg9433Y0b2OVVg9433Y0b2OVVg9        2643
Y_kc7RV54b71Y_kc7RV54b71Y_kc7RV5        2632
iRW58SX90Xc5iRW58SX90Xc5iRW58SX9        1835
eVY341a_76QReVY341a_76QReVY341a_        1299
3UVVZ8S4_V4a3UVVZ8S4_V4a3UVVZ8S4        1270
```

Fig -9: Bad algorithm of deciphering

The similarity of key could by studied using similariy expressed by the equation (2) code on Matlab :

$$sim(A,B) = \frac{A.*B}{norm(A).norm(B)} \tag{2}$$

The bit position will be decomposed and the similarity of the key and the next key will be represented in the figure 10. The legend of all figures will be resumed by the table 2.

Table -1: Table of figure number

Bit position	Figure number (Bloc)	Colors
1-4	1	Red-Green-Blue-Yellow
5-8	2	Red-Green-Blue-Yellow
9-12	3	Red-Green-Blue-Yellow
13-16	4	Red-Green-Blue-Yellow
17-20	5	Red-Green-Blue-Yellow
21-24	6	Red-Green-Blue-Yellow
25-28	7	Red-Green-Blue-Yellow
29-32	8	Red-Green-Blue-Yellow

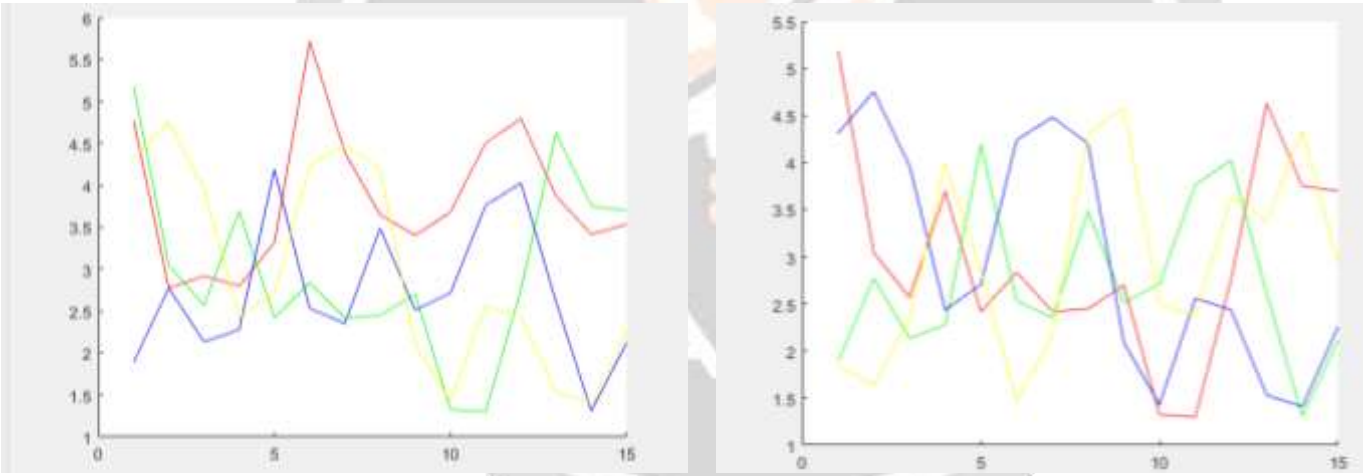

Fig -10: Bloc1 – Bloc2

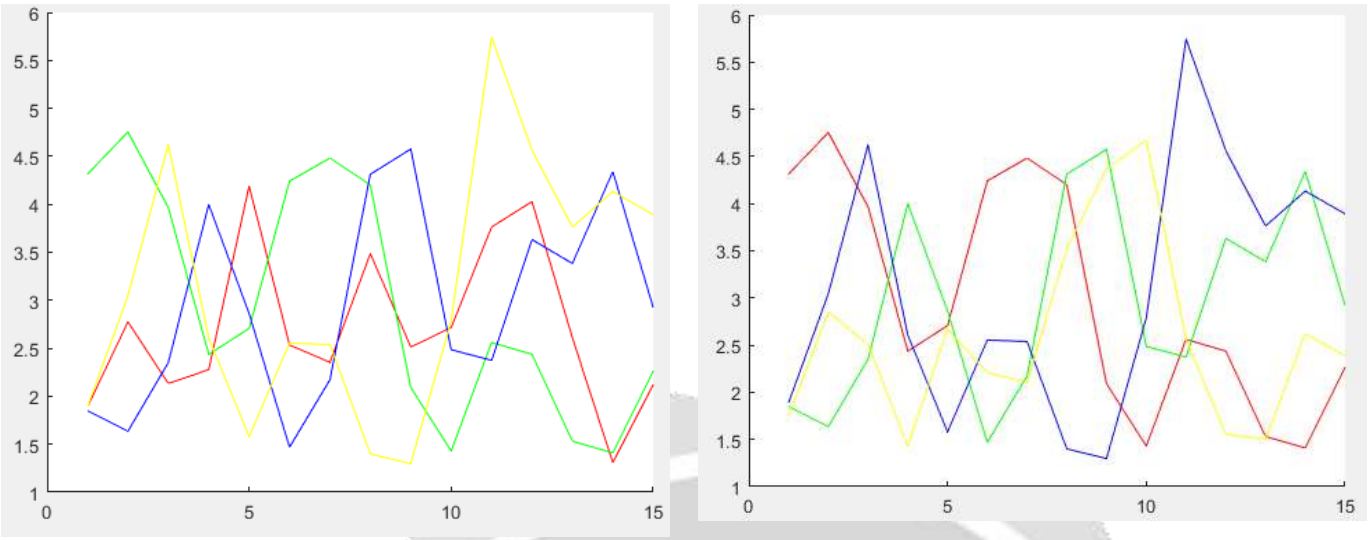

Fig -11: Bloc3 – Bloc4

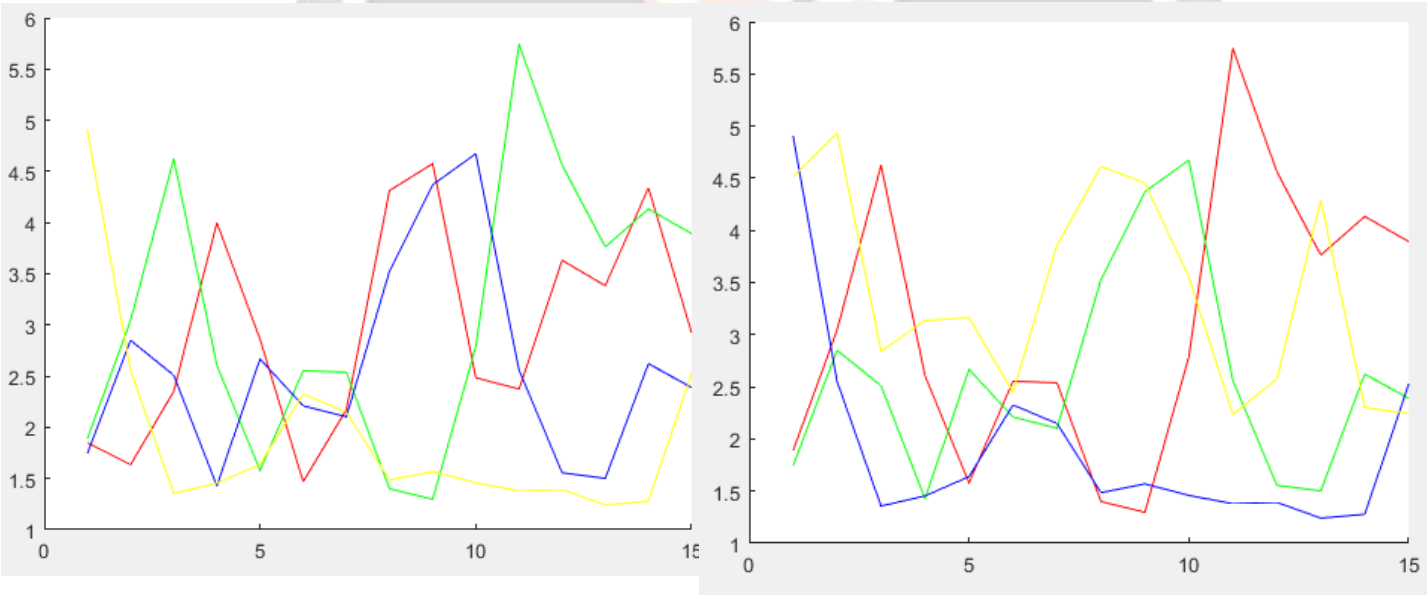

Fig -12: Bloc5 – Bloc6

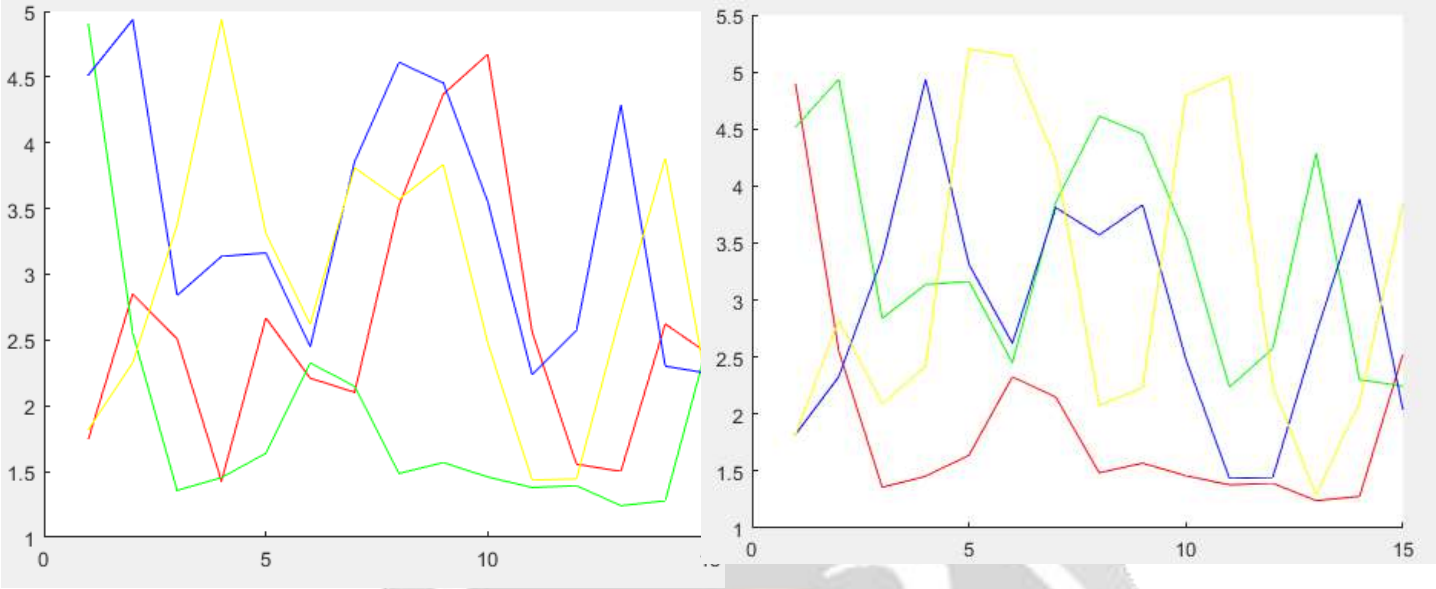

Fig -13: Bloc7– Bloc8

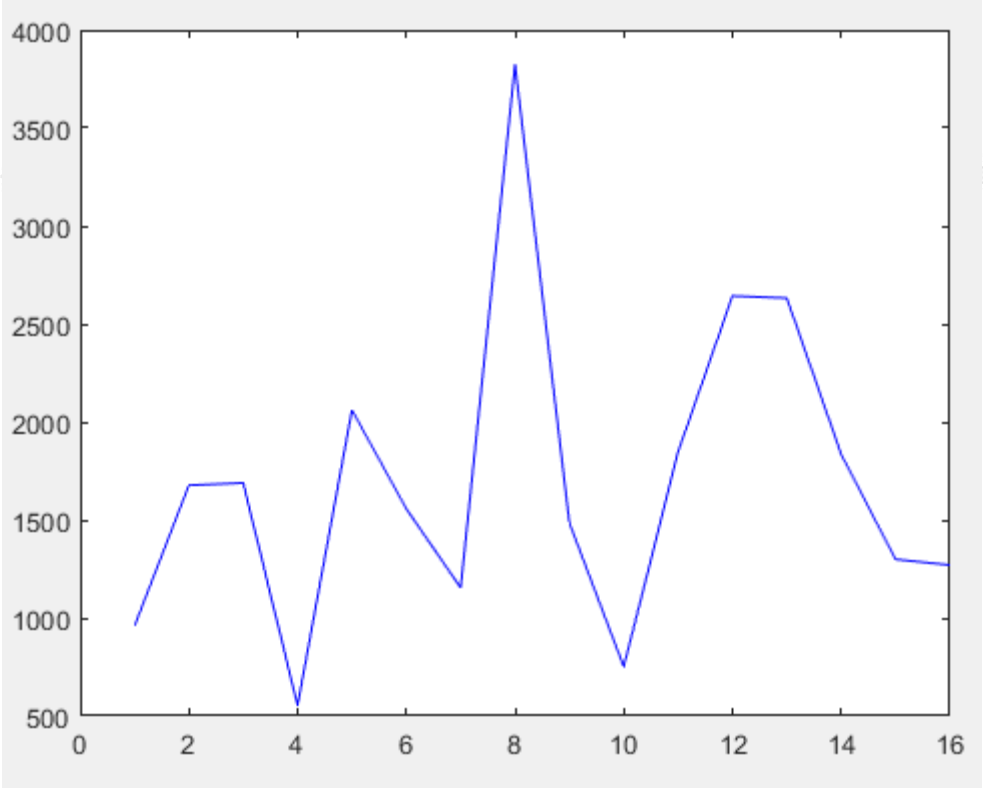

Fig -14: Evolution of iteration

Figures 10 until 13 represent the similarity of each key. We can see that similarity doesn't surpass the 6%. The key generated doesn't have a similarity between them. Figure 14 shows that the key is really dynamic and iteration changes for each experience.

3.4 Time execution

After some experience, the execution time increases quickly, but when the information doesn't change a lot, the ciphering is very quick. So, for text ciphering, adding compressive techniques will improve the result, like Run-Length Encoding (RLE) or Lempel-Ziv-Welch (LZW).

4. CONCLUSIONS

The neuronal cryptography could be used for generating the key of a mobile communication system with the similarity less than 6% for each key. The type of algorithm is also very important. So adding more symmetric key encryption with software will add more security to the text. The execution of time of compressed text is much quicker than normal text. As a perspective, adding RLE or LZW with this algorithm will be very beneficial.

5. REFERENCES

[1]. V. Gujral, S. Pradhan, "Cryptography using artificial neural networks", 2009 https://www.researchgate.net/publication/37394331
[2]. A. Jagadev, "Advanced Encryption Standard (AES) Implementation", M.Tech Thesis National Institute of Technology, Rourkela. May, 2009.
[3]. Andreas Ruttor, "Neural Synchronization and Cryptography ", PhD thesis, Bayerische Julius MaximilianUniversity Wurzburg, 2006
[4]. https://www.garykessler.net/library/crypto.html, consultation date : April 2022
[5]. https://www.cl.cam.ac.uk/~rja14/serpent.html, consultation date : April 2022