C++ Implementation of Neural Cryptography for Public Key Exchange and Secure Message Encryption with Rijndael Cipher

Sagun Man Singh Shrestha

Department of Electronics and Computer Engineering,
Tribhuvan University – Kathmandu Engineering College, Nepal

Google: sagunms | github.com/sagunms/NeuroCrypto

Abstract: This work is the software implementation of the concept of neural cryptography, which is a communication of two tree parity machines for agreement on a common key over a public channel. This key is utilized to encrypt a sensitive message to be transmitted over an insecure channel using Rijndael cipher. This is a new potential source for public key cryptography schemes which are not based on number theoretic functions, and have small time and memory complexities. This paper will give a brief introduction to artificial neural networks, cryptography and its types, which will help explain why the two communicating terminals converge to a common key in neural cryptography and will also cover the Rijndael (AES) cipher. This paper is intended to show that such neural key exchange protocol and AES encryption can be practically implemented in a high-level programming language viz. C++, which could be further extended in higher-level applications. Both CLI and GUI implementations of the software created using Visual C++ (.NET framework) are presented.

Index Terms: Neural Networks, Tree Parity Machine, Mutual learning, Cryptography, Public Key, Symmetric Key, Rijndael, Advanced Encryption Standard, Visual C++, .NET Framework

I. INTRODUCTION

To understand the underlying concepts of neural cryptography and Rijndael, one should first have proper perspective on artificial neural networks, cryptography and its types in general.

1.1 Artificial Neural Networks

Artificial Intelligence is the study of the computations that make it possible for computers to perceive, reason and act. An artificial neural network (ANN) is an information processing paradigm inspired by the structure and functional aspects of biological neural networks, such as the brain, to process information. The key element of this paradigm consists of highly interconnected processing elements called artificial neurons, working in unison to solve specific problems, just like our own brains do - learning by example.

Taking inspiration from the working of biological neurons, the mathematical model of an artificial neuron (Fig. 1) can be formulated. The inputs are multiplied by its corresponding random weight (constantly adjusted during learning), summed up, and if the end result exceeds the threshold value, then it fires an output which goes through activation function. It is these values of the weights (synaptic strengths) and the threshold value that is continuously adjusted until the optimum solution is found.

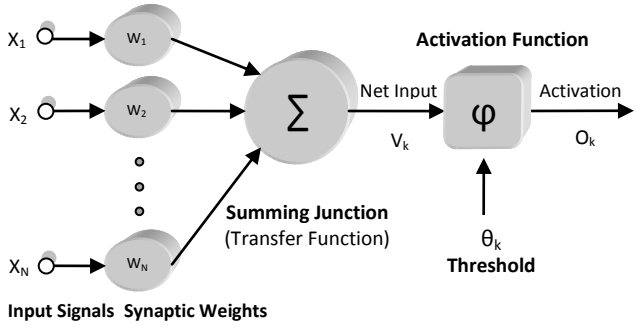

Figure 1: Model of a single artificial neuron

An artificial neural network (Fig. 2) is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. It is configured for a

specific application, such as pattern recognition or data classification, through a learning process. Just as learning in biological systems involves adjustments to the synaptic connections that exist between the neurons, this is true of ANNs as well.

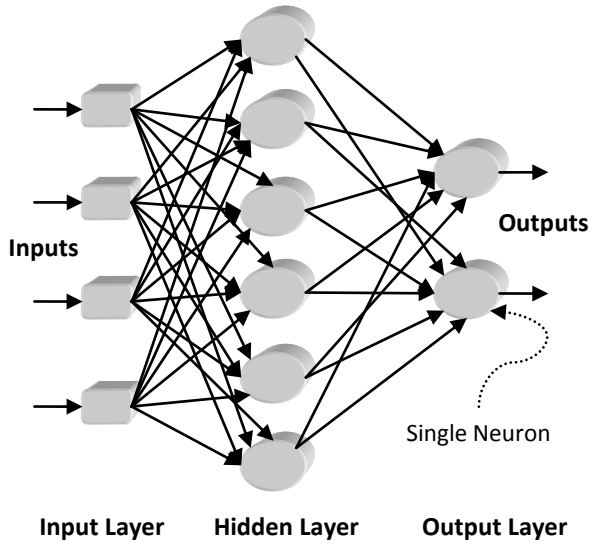

Figure 2: Model of an artificial neural network

1.2 Cryptography

Cryptography, as derived from Greek *kryptos*, "secret" and *gráph*, "writing", is the practice and study of hiding information. It intersects the disciplines of mathematics, computer science, and engineering. Applications include ATM cards, computer passwords, and e-commerce. *Encryption* is the process of converting ordinary information (*plaintext*) into unintelligible gibberish (*ciphertext*) while *decryption* is the reverse. A *cipher* is a pair of algorithms that create the encryption and the reversing decryption. The detailed operation of a cipher is controlled both by the algorithm and in each instance by a secret parameter called *key*. Keys are important, as ciphers without variable keys can be trivially broken with only the knowledge of the cipher used and are therefore useless. The modern field of cryptography can be divided into two broadly areas - Symmetric and Public-key cryptography.

1.2.1 Symmetric-key cryptography

This was the only kind of encryption publicly known until June 1976. Its modern study relates mainly to the study of *block ciphers* (e.g. DES, Rijndael) and *stream ciphers* (e.g. RC4) and to their applications. In this method, both the sender and the receiver use the same key for encryption and decryption of a message, though a message or group may have a different key than others. A disadvantage is the key management necessary to use them securely. Each distinct pair of communicants must share a different key and each ciphertext exchanged as well. The number of keys required increases as the square of the number of network members, which requires complex key management schemes to keep them all secret.

1.2.2 Public-key (Asymmetric) cryptography

In a groundbreaking 1976 paper, Whitfield Diffie and Martin Hellman proposed the notion of public-key cryptography in which two different but mathematically related keys are used - a *public* key and a *private* key. This was first practical method for establishing a shared secret over an unprotected communications channel. The system is so constructed that calculation of one key (*private key*) is computationally infeasible from the other (*public key*). Both keys are generated secretly, as an interrelated pair. An unpredictable large random number is used to begin generation of an acceptable pair of keys suitable for use by an asymmetric key algorithm. The *public key* may be freely distributed and typically used for encryption, while its paired *private key* must remain secret and used for decryption. Unlike symmetric key algorithms, it does not require a secure initial exchange of one or more secret keys to both sender and receiver. The critical advantage is that communicating parties (traditionally Alice and Bob) never need to send a copy of their keys to each other therefore even if say, Bob’s key is known to a third party, Alice's secret messages to Bob would be compromised, but Alice’s messages would still be secure to others.

2. NEURAL CRYPTOGRAPHY

It is a new branch of cryptography which incorporates neural networks with cryptography. It is dedicated to analyzing the application of stochastic algorithms namely, neural network algorithm, for use in either *cryptanalysis* or *encryption*. The first work that is known on this topic can be traced back to 1995 in an IT Master Thesis by Sebastien Dourlens. Due to its recent development, there hasn’t been much

practical applications as of yet. However, like any nascent field, rapid research findings on neural cryptography is bringing new and exciting ideas. It could be used specially where the keys are continually generated and the system (both pairs and the insecure media) is in a continuously evolving mode. ANNs are well known for their ability to selectively explore the solution space of a given problem. The ideas of mutual learning, self learning, and stochastic behavior of ANNs can also be alternatively used for the realization of symmetric key exchange protocols, mutual synchronization, and generation of pseudo-random numbers. With these features, two neural networks can be used to encrypt and decrypt or exchange messages in a network, which can be used for public-key *encryption*.

2.1 Neural key exchange protocol

The most used protocol for key exchange between two terminals in practice is Diffie-Hellman protocol. Neural key exchange, based on the synchronization of two TPMs, each associated with parties wishing to share a secret, should be a secure replacement for this method. Chaos theory studies the behavior of dynamical systems that are highly sensitive to initial conditions (aka. butterfly effect). Even small differences in initial conditions (like rounding errors in numerical computation) yield widely diverging outcomes for chaotic systems, rendering long-term prediction impossible. Therefore, synchronization of these two TPMs at A and B in neural cryptography has a striking similarity in synchronization of two chaotic oscillators in chaos communications.

2.2 Tree Parity Machines

The tree parity machine (Fig. 3) is a special type of multi-layer feed-forward neural network. It consists of one output neuron, K hidden neurons and K * N input neurons. Inputs to the network are binary.

$$x_{ij} \in \{-1, +1\}$$

Weights of input and hidden neurons take the values.

$$w_{ij} \in \{-\mathrm{L}, \ldots, 0, \ldots, +\mathrm{L}\}$$

Output of each hidden neuron is calculated as a sum of all products of input neurons and its weights.

$$\sigma_i = sgn\left(\sum_{j=1}^{N} w_{ij}\ x_{ij}\right)$$

Signum is a simple function, which returns -1, 0 or 1.

$$sgn(x) = \begin{cases} -1 & if\ x < 0 \\ 0 & if\ x = 0 \\ 1 & if\ x > 0 \end{cases}$$

If the scalar product is 0, the output of hidden neuron is mapped to -1 to ensure a binary output value. The output of neural network is then computed as the product of all values produced by hidden elements.

$$\tau = \prod_{i=1}^{K} \sigma_i$$

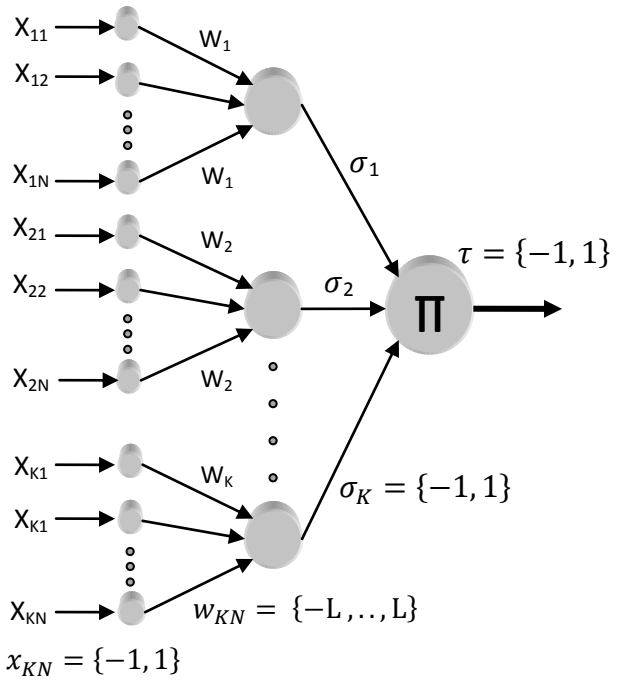

Figure 3: Structure of Tree Parity Machine

Summarizing, the output of the TPM is in binary and can be written as:

$$\tau = \prod_{i=1}^{K} sgn\left(\sum_{j=1}^{N} w_{ij}\ x_{ij}\right)$$

2.3 Feasibility and Security

The dynamics of two networks and their weight vectors found exhibit a novel phenomenon, where network synchronize to a state with identical time dependent weights. This concept of fast synchronization by mutual learning can be applied to secret key exchange protocol over a public channel and the generated key can be used for encryption and decryption of a given message. The algorithm does not operate on large numbers and methods from number theory and therefore leads to fast synchronization of the public key. The security of neural cryptography is still being debated but since the method is based on a stochastic process, there is a small chance that an attacker synchronizes to the key, as well. It has been found that the model parameter *L* determines the security of the system.

2.4 Synchronization Algorithm

Each terminal A and B uses its own tree parity machine. Synchronization of the tree parity machines is achieved in the following steps.

1. Initialize random weight values
2. Execute these steps until the full synchronization is achieved
    1. Generate random input vector X
    2. Compute the values of the hidden neurons
    3. Compute the value of the output neuron
    4. Compare the values of both tree parity machines
        1. If outputs are others: go to 2.1
        2. If outputs are same: one of the suitable learning rules is applied to the weights

The flowchart of *NeuroCrypto* as shown in Fig. 4 is the high-level programming language representation of neural cryptography. After the full synchronization is achieved (weights w_{ij} of both tree parity machines are the same), A and B can finally use their weights as keys. This method is known as *bidirectional learning*.

For synchronization, the weights of the TPM have to be constantly checked for equality and then updated. The weights are updated only if the output values of the two TPMs are equal. One of the following learning rules can be used for the synchronization:

- Hebbian learning rule:
  $w_{ij}^{+} = g\{ w_i + \sigma_i x_i \ominus (\sigma_i \tau) \ominus (\tau^A \tau^B) \}$
- Anti-Hebbian learning rule:
  $w_{ij}^{+} = g\{ w_i - \sigma_i x_i \ominus (\sigma_i \tau) \ominus (\tau^A \tau^B) \}$
- Random walk:
  $w_{ij}^{+} = g\{ w_i + x_i \ominus (\sigma_i \tau) \ominus (\tau^A \tau^B) \}$

Where, $\ominus$ (*Theta*) is the Heaviside step function, if the input is positive then the output is 1 and if input is negative then the function evaluates to 0.

Theta(a, b)=0 if *a<>b*; else *Theta=1*. The function g(…) keeps the weights in the range *{-L, +L}*. *x* is the input vector and *w* is the weights vector. After the machines are synchronized, their weights are equal: we can use them for constructing a shared *key*. There could be various possibilities that this algorithm could be attacked. However, the possibilities of this happening are very unlikely.

2.5 C++ Implementation (*NeuroCrypto* CLI)

2.5.1 *InputVector* class structure

```
class TPMInputVector {
    public:
    DynamicArray <int> X;
    void CreateRandomVector(int K,int N);
    void xLength (int K, int N);
};
```

This class dynamically allocates and updates the randomized input vector *X* which will be constantly accessed by *NeuroCrypto* class for updating the weights during synchronization.

The input vector *X* is used throughout the synchronization process to randomly assign weights of the neural network. The two functions of *TPMInputVector* class are *CreateRandomVector* and *xLength*. *CreateRandomVector* assigns random signed bits (-1 or 1) to all the `K*N` number of neurons while, *xLength* is used to allocate the `K*N` long dynamic array *X* to store the input vector.

2.5.2 *TreeParityMachine* class structure

```
class TreeParityMachine {
    public:
    DynamicArray <int> W, H;
    int K, N, L;
    int TPMOutput;
    void Initialize ();
    void ComputeTPMResult (const
                DynamicArray <int> &X);
    void UpdateWeight (const
                DynamicArray <int> &X);
    void RandomWeight ();
};
```

This is the TPM class which is based on the mathematics and algorithm covered in Listings 2.2 and 2.4 and is composed of weight and hidden intermediate *DynamicArray* objects, *K*, *N* and *L* parameters to characterize TPM and a TPM output variable. Various functions for initialization, weight updating and result computation are defined.

2.5.3 *Tree Parity Machine* class definitions

```
TreeParityMachine::ComputeTPMResult
        (const DynamicArray <int> & X) {
    int i, j, sum;    TPMOutput = 1;
    for (i = 0; i < K; i++) {
        sum = 0;
        for (j = 0; j < N; j++)
            sum+=(W.Z[i*N+j] * X.Z[i*N+j]);
        H.Z[i]= Signum(sum);
        TPMOutput *= Signum (sum);
    }
}
void TreeParityMachine::Initialize() {
    W.length(K * N);  H.length(K);
}
void TreeParityMachine::RandomWeight(){
    int i;
    for (i=0; i < K*N; i++)
        W.Z[i] = L-(rand()%(2*L+1));
}
void TreeParityMachine::UpdateWeight
        (const DynamicArray <int> & X) {
    int i, j, newW;
    for (i = 0; i < K; i++) {
        for (j = 0; j < N; j++) {
            newW = W.Z[i * N + j];
            newW += X.Z[i * N + j] *
TPMOutput*IsEqual(TPMOutput,H.Z[i])
*IsEqual(TPMOutput, TPMOutput);
            if (newW > L)     newW = L;
            if (newW < - L)   newW = -L;
            W.Z[i * N + j] = newW; }
    }
}
```

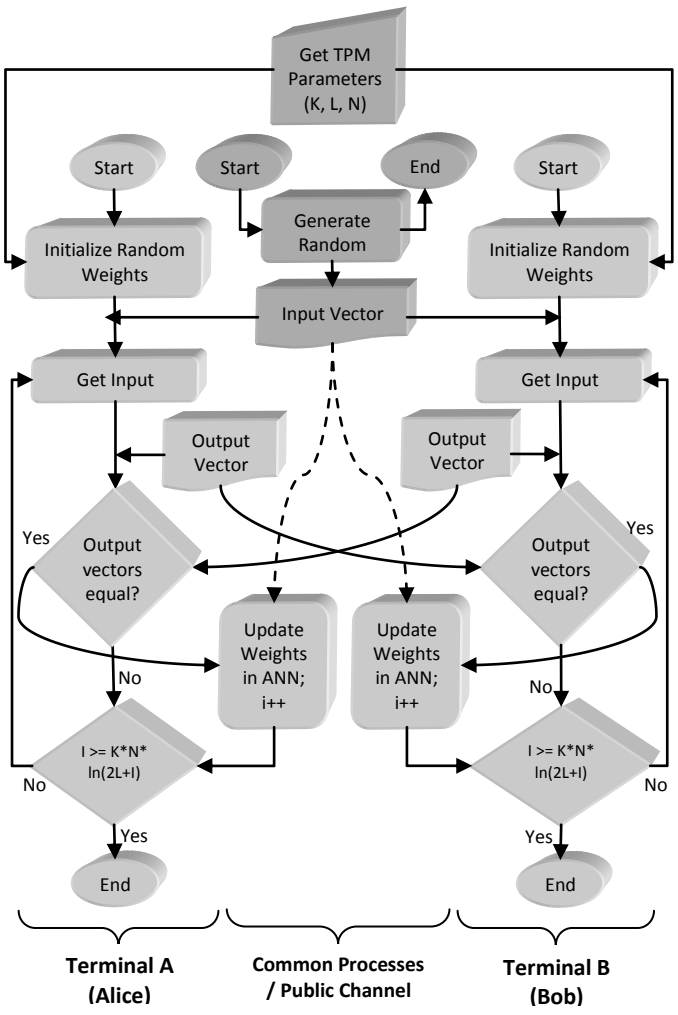

Figure 4: Overall program flowchart of High-level language implementation of Neural Cryptography

2.5.4 *NeuroCrypto* class structure

```
class NeuroCrypto {
    public:
    unsigned int itrMax;
    TreeParityMachine A, B;
    TPMInputVector objInput;
    char publickey[100];
    //Default Constructor
    NeuroCrypto (int k,int n,int l);
};
```

This is the superset class of *NeuroCrypto* program module which consists of two *TreeParityMachine* objects *A* and *B* for Alice and Bob's TPM, a *TPMInputVector* object *objInput*, a character array to store the final public key. The default constructor takes the common TPM parameters *K*, *N* and *L* inputted by the user during runtime.

2.5.5 Global Function Declaration

```
int IsEqual (int A, int B);
int RandomBit ();
int Signum (double r);
```

Global functions for checking equality, generating random bit (either 1 or -1) and Signum are declared.

2.5.6 Variable Declarations and Initialization

```
int i, ii, ss, itrMax, j,K=0,sum,
       key_size,key_length,initN,initL;
TreeParityMachine A, B;
TPMInputVector objInput;
DynamicArray <char> publickey;
const char Dictionary [38] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789";
srand (time(NULL)); //random generator
cout << "Parameter settings (K, N, L)";
cin >> initK >> initN >> initL;
A.K=initK, A.N=initN, A.L=initL;//InitA
A.Initialize (); A.RandomWeight ();
B.K=initK, B.N=initN, B.L=initL;//InitB
B.Initialize (); B.RandomWeight ();
itrMax=(A.L*A.L*A.L*A.L)*A.N*A.K; /////
cout << "Maximum Iterations: "<<itrMax;
objInput.xLength (B.K, B.N);
cout<<"Synchronizing TPM Networks...";
```

A and B (TPMs of Alice and Bob) are instances of *TreeParityMachine* class. The *publickey* object of *DynamicArray* class stores the final publickey after successful sync. *Dictionary* array stores 38 symbols (26 alphabets, 10 numbers, and an underscore) as a template for key generation. The system time is used as a seed to generate random numbers. The program takes the parameters *K*, *N* and *L* from the user and initializes the TPMs with the common parameters.

2.5.7 Main Iteration

```
for (i=1; i!=itrMax; i++) {
    objInput.CreateRandomVector(B.K, B.N);
    A.ComputeTPMResult(objInput.X);
    B.ComputeTPMResult(objInput.X);
    if(A.TPMOutput == B.TPMOutput){
        A.UpdateWeight (objInput.X);
        B.UpdateWeight (objInput.X);
        sum = 0;
        for(ss=0;ss<A.K*A.N;ss++)//Find sum
        sum += abs(A.W.Z[ss]-B.W.Z[ss]);
        if ( sum == 0 ) break;      } }
if (sum == 0) cout << "Status: SUCCESS!";
else cout << "Status: FAILED!";
```

For each iteration of Alice's TPM, random input vector will be produced (using *CreateRandomVector* function of *TPMInputVector* class), the output value of TPM will be computed (using *ComputeTPMResult* function) and this output value will be available to Bob's TPM B. Bob's TPM should follow the same iterative procedure. During synchronization, both parties A and B continually check for the *equality* of both their *TPMOutput* values.

2.5.8 Key Generation and Sync Outputs

```
cout<<"Iterations:"<< i <<"DataExchanged:"
     << (i*(A.K*A.N+4)/1024) << " KiB";
key_size = 37 / (A.L * 2 + 1);
key_length = A.K * A.N / key_size;

cout << "Key length: " << key_length;
publickey.length(key_length + 1);

for(i = 0; i < key_length; i++)
    publickey.Z[i] = 0;
for (i=1; i < key_length+1; i++) {
    K = 1;
    for(j=(i-1)*key_size; j<i*key_size;j++)
        K = K + A.W.Z[j] + A.L;
    //
    publickey.Z[i-1]=Dictionary[K];
}
publickey.Z[key_length]='\0'; //Null char
cout << "Public Key: " << publickey.Z;
```

The actual iterations that were required to accomplish TPMs of matching weights are recorded in *i*. For each iteration, the data transferred is `K*N+4` bytes. So for whole synchronization, it would be `i*(K*N+4)/1024` kilobytes. The length of the public key is calculated as the total number of neurons of TPM divided by the key size, where again key size depends on the number of symbols assigned in the dictionary and the depth of weights in both within the range *{-L, L}*. This explains the formula `L*2+1` (to include origin). For publickey generation of *key_length* characters, the i^{th} character of *publickey* will be assigned as the k^{th} symbol in the dictionary. This will be done such that the location *K* depends on j^{th} neural weight, which again depends on the i^{th} position of *publickey* array. This iterative formula for key generation is done by a nested loop to produce a seemingly random public key which actually depends on the weights, dictionary size, and the TPM parameters (depends on *key_size*).

3. RIJNDAEL (AES)

Advanced Encryption Standard (AES) is a symmetric-key encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as *Rijndael* (coined after two Belgian cryptographers, Vincent *Rij*men and Joan *Dae*men). The AES ciphers have been analyzed extensively and are now used worldwide, as the case with its predecessor, the DES.

AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits, whereas Rijndael can be specified with block and key sizes in any multiple of 32 bits, with a minimum of 128 bits. The block size has a maximum of 256 bits, but the key size has no theoretical maximum. AES operates on a 4×4 array of bytes, termed the *state* (versions of Rijndael with a larger block size have additional columns in the state).

The cipher is specified as a number of repetitions of transformation rounds that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key – the public key obtained from the synchronization of two TPMs during the neural cryptography stage. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

3.1 High Level Algorithm

1. *KeyExpansion*: round keys are derived from the cipher key using Rijndael's key schedule
2. Initial Round
    1. *AddRoundKey*: each byte of the state is combined with the round key using bitwise XOR
3. Rounds
    1. *SubBytes*: a non-linear substitution step where each byte is replaced with another according to a lookup table.
    2. *ShiftRows*: a transposition step where each row of the state is shifted cyclically a certain number of steps.
    3. *MixColumns*: a mixing operation which operates on the columns of the state, combining the four bytes in each column.
    4. *AddRoundKey*
4. Final Round (no *MixColumns*)
    1. *SubBytes*
    2. *ShiftRows*
    3. *AddRoundKey*

3.2 Security and Feasibility

Until May 2009, the only successful published attacks against the full AES were side-channel attacks on some specific implementations. The National Security Agency (NSA) reviewed Rijndael, and stated it was secure enough for U.S. Government non-classified data and that design and strength of all key lengths of the AES (128, 192 and 256) are sufficient to protect classified information up to the 'Secret' level. 'Top Secret' will require use of either 192 or 256 key lengths. Thus, the use of Rijndael for encrypting sensitive messages using the synchronized public key as used in this program could provide a very strong level of security. The demonstration software: *NeuroCrypto* uses the *Cryptostream* class of *System::Security::Cryptography* namespace of Visual Studio .NET library to implement the 128 bit key size, ECB mode Rijndael cipher. Since it can be configured to use any variant of Rijndael (including AES), the flexible term *Rijndael* is used.

4. CONCLUSION

The difficulty of securely establishing a secret key between two communicating parties, when a secure channel does not already exist between them is a considerable obstacle for using only a symmetric cipher such as Rijndael. There are several attractive features of neural cryptography such as fast synchronization of a common key and its striking similarity with the synchronization of two chaotic oscillators which gives a very small chance for the eavesdropper to synchronize its TPM with the two TPMs of the communicating parties. Taking these facts into consideration, a hybrid cascaded system consisting of both neural cryptography and Rijndael cipher was created using Visual C++ .NET in both CLI and GUI versions as shown in the screenshot (Listing 7). Exploiting the advantages from both the ciphers, this could be a promising architecture for a stronger and more secure cryptography system that could be used for various applications of data security over insecure channels.

5. FUTURE SCOPE

This project: *NeuroCrypto* is just a proof-of-concept demonstration to show that a strong cryptographic system using neural cryptography cascaded with Rijndael cipher can be implemented in using a high-level language. It still has room for a lot of improvement. It is released as open source under Apache License v2 at *github.com* so anyone can easily integrate this in higher-level software to share sensitive data over the internet or any insecure channel, provided, the channel is feasible for rapid synchronization of both TPMs. Good performance (high speed and low RAM requirements) were an explicit goal of the AES selection process. Thus AES performs well on a wide variety of hardware, from 8-bit microcontrollers to high-performance computers. So this software could also be optimized to work on embedded platforms in applications such as Ethernet-based or wireless systems. Therefore, this project can be expanded to cover multiple cryptography domains.

6. REFERENCES

[1] N. Prabakaran, P. Saravanan, and P. Vivekanandan, 2008. "A New Technique on Neural Cryptography with Securing of Electronic Medical Records in Telemedicine System", International Journal of Soft Computing 3 (5): 390-396, Medwell Journals, 2008.

[2] Andreas Ruttor, Ph. D. Thesis, "*Neural Synchronization and Cryptography"*, Bayerische Julius-Maximilians-Universität Würzburg, 2006

[3] CyberTrone, 2009. http://codeproject.com/Articles/39067/Neural-Cryptography

[4] E. Klein, R. Mislovaty, I. *Kanter*, W. Kinzel; "*Synchronization of neural networks by mutual learning and its application to cryptography"*.

[5] A. Klimov, A. Mityaguine, A. Shamir; "*Analysis of Neural Cryptography"*, Advances in Cryptology, ASIACRYPT 2002.

[6] P. Revankar, W. Z. Gandhare and D. Rathod, 2010. *"Private Inputs to Tree Parity Machine"*, International Journal of Computer Theory and Engineering, Vol.2, No.4,Aug 2010,pp. 665-669.

[7] *Neural Cryptography*, *Cryptography*, *Neural Networks*, *Chaos Theory*, *Advanced Encryption Standard*. http://en.wikipedia.org

[8] S. Trenholme, 2005. "*AES Galois field*", "*Rijndael's key schedule*", "*Mix column stage*", "*AES encryption*". http://www.samiam.org

7. SCREENSHOTS (NeuroCrypto CLI/GUI)