https://doi.org/10.0000/tmsce.2026.001



# Towards a Secure Key Exchange Protocol Based on Artificial Neural Networks

Shalini Roy[1] and Somnath Kumar[2]

[1]Data Analytics Department, Life Card, Tokyo, Japan
[2]National Institute of Technology Jamshedpur, Jharkhand, India
*Corresponding author:* somnath1691997@gmail.com

**Abstract.** The work presents a correlation and connection between artificial neural networks and cryptography. We explore in detail how artificial neural networks can be employed to build secure key exchange protocols. We show how neural networks like *tree parity machine* and *permutation parity machine* can be utilized for the development of a secure key exchange protocol. We discuss different types of learning rules like *Hebbian*, *Anti-Hebbian*, and *Random walk* to achieve the synchronization of the neural networks. The neural key exchange protocol is based on the distinction between unidirectional and bidirectional neural network synchronization, instead of relying on number-theoretic principles. We also propose a multi-key neural network based on a permutation parity machine. In the following, we present a multi-key exchange protocol based on our proposed neural network.

**Key words:** Cryptography; Neural Networks; Key Exchange
**Received:** 1 January 2026 **Revised:** 26 January 2026 **Accepted:** 27 January 2026

## 1. Introduction

Constructing a secure communication channel is regarded as a highly demanding area of research in contemporary communication. Designing secure channels for communication is a complex and emerging research area in contemporary communication. The demand for secure, effective, and fast transmission protocols is crucial for various applications like mobile phones, satellite, and internet-based communications [1,2]. To achieve secure communication, a cryptosystem is needed that enables two parties to communicate without an opponent understanding and decrypting the transmitted message. Typically, this is done through a key-based encryption system. In a private-key system [3, 4], the communicating parties share a secret key prior to message transmission. In contrast, in a public-key system [3, 4], the encryption key is published, and a hidden communication is not necessary. However, an opponent cannot decrypt the message without knowing the decryption key. A *key exchange protocol* [3] is a cryptographic mechanism that allows two communicating parties to establish a common secret key for enabling secure communication. These protocols serve an essential role in the privacy of digital communications, preserving security, and are extensively deployed in applications such as online banking, electronic commerce, and secure messaging. Widely adopted key exchange mechanisms include the Diffie–Hellman key exchange (DHKE) [5,6], the Elliptic Curve Diffie–Hellman key exchange (ECDH) [7, 8], and RSA-based key exchange [9] among

others. Fundamentally, the security guarantees of these schemes is rooted in arember theoretic hardness assumptions [10–12] like integer factorization and discrete logarithm.

Classical cryptography-based key exchange protocols may not be safe in the future because of quantum computers. Many classical key exchange methods, such as Diffie–Hellman key exchange (DHKE) and RSA, are based on number-theoretic problems. These problems are hard for classical computers, but they can be solved efficiently by quantum computers.

Quantum computers are capable of solving certain mathematical problems much faster than classical computers. In particular, Shor's algorithm [13, 14] allows efficient solutions to problems such as integer factorization and discrete logarithms. As a result, a sufficiently powerful quantum computer can break many classical key exchange schemes and recover secret information transmitted over public channels. This poses a serious threat to the security of traditional cryptographic systems.

To address this challenge, several post-quantum key exchange protocols have been proposed [15]. These protocols aim to provide security even in the presence of practical quantum computers. Ongoing research focuses on analyzing the security of these schemes and evaluating their efficiency for deployment in real-world applications. In this context, neural cryptography has gained attention as a potential post-quantum approach, where neural networks are employed to perform key exchange operations [16–20].

Neural cryptography differs fundamentally from classical cryptographic techniques. While classical cryptography mainly relies on number-theoretic problems and public-key constructions, neural cryptography is based on the learning behavior of artificial neural networks. In this approach, two neural networks interact and learn simultaneously until they reach an identical internal state. This learning and synchronization process is used to establish a shared secret key over a public communication channel.

In neural cryptography, two users first decide to use the same neural network. This network can be a Tree Parity Machine or a Permutation Parity Machine. Both users build their own network using the same public values. However, they choose the starting weights randomly.

During the key exchange process, the same random input is given to both networks. After that, the users share only the output of their networks with each other. They do not share the weights. Using these output values, each network changes its weights by following a learning rule. This process is done again and again. After some time, both networks reach the same weight values. When this happens, the networks are said to be synchronized. These same weight values are used as the secret key.

After synchronization, the shared weights are treated as a secret key. This key can be used for many security purposes. For example, it can be used to encrypt and decrypt messages, to check data correctness, and for user authentication.

Neural cryptography is secure because it is very hard for an attacker to copy this learning process. Even if an attacker can see all the inputs and outputs sent on the public channel, it is still difficult to get the same weights. This is because the starting weights and inputs are random. Also, the learning rule updates weights only in some cases. Due to these reasons, an attacker cannot easily guess or recreate the secret key.

## 2. Preliminaries

### 2.1. Diffie-Hellman Key Exchange

The Diffie–Hellman key exchange is a method that helps two users create a secret key. This key is shared over a public channel. Even if someone else is listening, they cannot easily find the secret key. Diffie and Hellman introduced this idea in the year 1976 [5]. Since then, this

method has become very important and is widely used in many modern security systems. The key exchange works by using mathematical operations on large numbers. The two parties are participating in the exchange; each chooses a random number and performs a series of calculations to arrive at a shared secret value. After the successful execution of the key exchange protocol, both parties obtain a common shared secret that is never revealed over the public channel. Since the computed shared secret is identical for the communicating parties, it is suitable for use as a symmetric encryption key. The protocol works as follows:

- Let the two communicating entities, Dora and Smith, publicly select a large prime number $p$ and a corresponding base $\ell$, such that $\ell$ is a primitive root modulo $p$.
- Dora chooses a secret random number $\mathbf{d}$ and calculates $D = \ell^{\mathbf{d}} \mod p$. She sends $D$ to Smith.
- Smith chooses a secret random number $\mathbf{s}$ and calculates $S = \ell^{\mathbf{s}} \mod p$. He sends $S$ to Dora.
- Dora computes the shared secret key as $K = S^{\mathbf{d}} \mod p$.
- Smith computes the same shared secret key as $K = D^{\mathbf{s}} \mod p$.

As a result, both Dora and Smith obtain an identical shared secret $K$, which can be employed as the key for a symmetric encryption scheme to secure their communication. An adversary who intercepts the exchanged messages cannot recover the shared secret, since it is never revealed and remains known only to the legitimate parties. The security of this key exchange mechanism relies on the computational hardness of the discrete logarithm problem over finite fields.

### 2.2. Machine Learning

Artificial intelligence (AI) [21] is a field of research that focuses on building computer systems that are able to learn from data and perform tasks that usually require human intelligence. Such tasks include problem solving, decision making, and perception. In AI, different algorithms, machine learning methods, and statistical models are used so that computers can imitate certain human cognitive abilities and work independently without continuous human involvement.

Machine learning is a part of artificial intelligence. In machine learning, computers learn from data. They do not follow fixed rules written by humans. First, the machine is given a lot of data. By seeing this data again and again, the machine learns how to make decisions or predictions. It understands things by finding simple patterns in the data. When more data is given, the machine becomes better and more accurate. What the machine learns is saved, and later it uses this knowledge to work on new data.

The first one is supervised learning. In this type, the machine is given data along with the correct answers. The machine checks its output and compares it with the correct answer. By doing this again and again, it learns how to give the right result. After training, it can also give answers for new data.

The second one is unsupervised learning. In this type, the machine is given only data and no answers. It tries to understand the data on its own. It looks for similarities and differences and then makes groups of similar data.

The third one is reinforcement learning. In this type, the machine learns by trying different actions. If it does something right, it gets a reward. If it does something wrong, it gets a

punishment. By repeating this process many times, the machine slowly learns how to make better decisions.

### 2.3. Artificial Neural Network

An Artificial Neural Network, or ANN, is a simple learning model used in machine learning. It is made by copying the basic idea of how the human brain works. An ANN has many small parts called neurons. These neurons are connected to each other and work together to handle information.

An ANN is arranged in different layers. The first layer is called the input layer. This layer receives data in numbers and sends it inside the network. The last layer is known as the output layer. It gives the final answer of the network. Between these two layers, there can be one or more hidden layers. These hidden layers help the network learn useful things from the input data and understand it better.

In the hidden layer, each neuron takes input from neurons of the previous layer. Every input value is multiplied by a weight, and then all values are added together. After that, this total value is passed through an activation function. The result of this function is then sent to the next layer.

Neurons in an ANN are connected through weights. These weights decide how much importance a connection has. At the start, the weights are selected randomly. During the training process, these weights are slowly adjusted. If the output given by the network is not correct, an error is calculated. This error is sent backward through the network using a method called backpropagation. Based on this error, the weights are updated so that the network gives better results in the future.

### 2.4. Neural Cryptography

Neural cryptography is a new and developing area of cryptography. It uses artificial neural networks to do basic security tasks. These tasks include making messages secret, reading secret messages, and creating a secret key.

In neural cryptography, two neural networks learn to communicate safely with each other. This communication happens even when the channel is not secure. One network acts like a sender. It takes a normal message and changes it into a secret message. The other network acts like a receiver. It takes the secret message and changes it back into the original message.

Both networks are trained at the same time so that they learn the same secret key. This key is learned using common inputs and outputs during training. Once the key is created, both networks use it to communicate in a secure way. Using this training data, the networks learn how to encode and decode messages with the shared key. Neural cryptography has the benefit of enabling secure communication channels to be established without the need for a complicated public key infrastructure. Instead, a secure key exchange protocol can be employed to generate a secret key shared by both parties.

A Tree Parity Machine with identical structures is utilized by partners $\mathcal{A}$ and $\mathcal{B}$. In the proposed setting, $\mathcal{K}$, $\mathcal{L}$, and $\mathcal{N}$ are disclosed publicly, whereas each neural network starts with randomly selected weight vectors that are preserved as secret initial conditions.

During the synchronization phase, the information exchanged over the public channel is limited to the input vectors $x_i$ and the overall outputs $\Gamma_{\mathcal{A}}$, $\Gamma_B$. As a result, each participant has access only to the internal representation $(\gamma_1, \gamma_2, \ldots, \gamma_K)$ of their own Tree Parity Machine. Preserving the confidentiality of this internal representation is crucial for the key exchange protocol to remain secure. Once full synchronization is reached, $\mathcal{A}$ and $\mathcal{B}$ utilize their aligned weight vectors as a shared secret key. Attacker $\mathcal{E}$'s primary challenge is the lack of knowledge

regarding the internal representations $(\gamma_1, \gamma_2, \ldots, \gamma_K)$ of $\mathcal{A}$'s and $\mathcal{B}$'s tree parity machines. Since the behavior of the weights relies on $\gamma_i$, correctly inferring the configuration of the hidden units is critical to launching a successful attack. It remains possible that a shrewd attack technique could be developed that completely undermines the security of neural cryptography. Nevertheless, such a risk is unavoidable in most cryptographic algorithms, as the one-time pad remains the only provably secure exception.

### 2.5. Neural Network Key Exchange

Neural Network Key Exchange (NNKE) is a protocol in which two parties, utilizes pre-trained neural networks to generate a shared secret key. The NNKE protocol involves both parties possessing a pre-trained neural network that can map input values to output values. The exchange of these output values between the parties is then utilized to obtain a shared secret key.

The NNKE protocol is executed as outlined below:

1. Two parties, say Dora and Smith, have pre-trained neural networks such as TPM that can map input values to output values.
2. Random weight values are initialized for both Dora's and Smith's neural networks.
3. The following steps are executed until full synchronization is achieved:
   (a) A randomly generated input vector $X$ is shared between the two parties, Dora and Smith.
   (b) Evaluate the hidden layer outputs of Dora and Smith.
   (c) Then the values of the output neuron $\Gamma^A$ and $\Gamma^B$ are calculated for Dora and Smith respectively.
   (d) Following that, the values of Dora's and Smith's Tree Parity machines are shared and compared.
   (e) If both parties obtain identical outputs, a suitable learning rule is used to modify the weight vectors.
   (f) Or else if the outputs are not equal, then we repeat the above steps, starting again from Step 1, until equal weights are achieved by both Dora and Smith.
4. After complete synchronization is attained—that is, when the weight vectors of both TPMs coincide—Dora and Smith can use these weights as the cryptographic key.

## 3. Neural Networks and Learning Rules

### 3.1. Tree Parity Machine (TPM)

The Tree Parity Machine (TPM) is a special type of neural network. It is commonly used in neural cryptography. Both the communicating users and attackers can use this network. The TPM is a multi-layer neural network that works in a feedforward manner. The TPM has $K$ hidden units. Each hidden unit works like a simple perceptron. Every hidden unit is connected to its own set of input neurons. Each hidden unit receives inputs from $N$ input neurons and produces one output value. The input values given to the TPM are discrete. These inputs can take only three possible values:

$$x_{ij} \in \{+1, 0, -1\}$$

The weights, which govern the transformation from input signals to output responses, are discrete-valued parameters restricted to the range between $-L$ and $+L$.

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

In this notation, the index $i = 1, \ldots, K$ refers to the $i^{th}$ hidden unit of the Tree Parity Machine, while $j = 1, \ldots, N$ indexes the individual components of the input vector. The output of each hidden neuron is obtained by computing the weighted sum of its input neurons, that is, by summing the products of the corresponding inputs and weights.

$$\gamma_i = sgn(\Sigma_{j=1}^{N} w_{ij} x_{ij})$$

The signum function is a simple mathematical operation that produces an output of $-1, 0, 1$, when the input value is negative, zero, or positive, respectively.

If the scalar product equals zero, the output of the corresponding hidden neuron is mapped to $-1$ to guarantee a binary output. The overall output $\Gamma$ of the Tree Parity Machine is then computed as the product (parity) of the outputs of all hidden units.

$$\Gamma = \prod_{i=1}^{K} \gamma_i$$

The output of the tree parity machine is binary.

### 3.2. Permutation Parity Machine (PPM)

The Permutation Parity Machine (PPM) is a binary version of the Tree Parity Machine. It is used as a practical option in neural cryptography. The PPM has three layers: an input layer, a hidden layer, and an output layer. The number of output neurons is based on the number of hidden units $K$. Inputs to the network take binary values:

$$x_{ij} \in \{+1, 0\}$$

The weights between input and hidden neurons are also binary:

$$w_{ij} \in \{+1, 0\}$$

Each hidden neuron's output value is determined through the summation of the XOR operations performed between each input neuron and its associated weight:

$$\gamma_i = \theta_N(\Sigma_{j=1}^{N} w_{ij} \bigoplus x_{ij})$$

$\theta_N(x)$ is a threshold function defined as follows:

$$\theta_N(x) = \begin{cases} 0 & x \leq N/2 \\ 1 & x > N/2 \end{cases}$$

The Permutation Parity Machine produces its total output $\Gamma$ by applying the exclusive-or (XOR) operation to the values of the hidden elements:

$$\Gamma = \bigoplus_{i=1}^{K} \gamma_i$$

```
def hebbian_rule(W,X,gamma,Gamma1,Gamma2,l):
for i in range(k):
for j in range(n):
W[i,j]+=X[i,j]*Gamma1* theta(gamma[i],Gamma1)*theta(Gamma1,Gamma2)
W[i,j]=np.clip(W[i,j],-l,l)
```

Figure 1: Hebbian learning rule in Python

## 4. Learning Rules in Neural Network

In this section, we talk about different learning rules in neural networks. Learning rules are very important. They tell the network how to change the weights during training. When the network gives a wrong output, the learning rule changes the weights. This helps the network get better next time.

In neural cryptography, learning rules are very useful. They help the network make a shared secret key. The rules make sure that both parties reach the same key. A person who is not allowed cannot easily guess or get this key. During training, the network changes the weights little by little. This makes the secret key stronger and safer. It also makes it harder for attackers to break the system.

### 4.1. Hebbian Learning Rule

The Hebbian learning rule is one of the oldest learning rules in neural networks. It was given by Donald Hebb in 1949. This rule explains how neurons learn together. If two neurons work at the same time, the connection between them becomes strong. If they do not work together, the connection becomes weak.

In neural cryptography, the Hebbian learning rule is used to change the connections between neurons that create the secret key. When the neurons give the same or similar output, their connections are increased. Because of this process, the neural network slowly learns the secret key. This secret key is hard for an unauthorized person to guess.

The mathematical form of the Hebbian learning rule is given as follows:

$$w_i^+ = g(w_i + \gamma_i x_i \Theta(\gamma_i \Gamma)\Theta(\Gamma^A \Gamma^B))$$

### 4.2. Anti-Hebbian Learning Rule

Anti-Hebbian learning is used in neural cryptography to make the secret key more unique and hard to guess. In this learning rule, the network changes its weights in a different way compared to the Hebbian rule. The secret key is created based on how the two neural networks behave during the training process. Because of this, an attacker cannot easily predict or copy the secret key. During training, both neural networks learn step by step. After training is completed, the two networks produce the same output when the same input is given. These same output values are then treated as the shared secret key. This secret key can be used for the encryption and decryption of messages between the two parties. The mathematical form of the Anti-Hebbian learning rule is given as follows:

$$w_i^+ = g(w_i - \gamma_i x_i \Theta(\gamma_i \Gamma)\Theta(\Gamma^A \Gamma^B))$$

```
def anti_hebbian_rule(W,X,gamma,Gamma1,Gamma2,l):
for i in range(k):
for j in range(n):
W[i,j]-=X[i,j]*Gamma1* theta(gamma[i],Gamma1)*theta(Gamma1,Gamma2)
W[i,j]=np.clip(W[i,j],-l,l)
```

Figure 2: Anti-Hebbian learning rule in Python

```
def randomwalk_rule(W,X,gamma,Gamma1,Gamma2,l):
for i in range(k):
for j in range(n):
W[i,j]+=X[i,j]* theta(gamma[i],Gamma1)*theta(Gamma1,Gamma2)
W[i,j]=np.clip(W[i,j],-l,l)
```

Figure 3: Random walk learning rule in Python

### 4.3. Random Walk

The Random Walk learning rule's basic concept is to adjust the weights of the network's neurons in accordance with how similar the inputs and outputs are. Specifically, the weights are updated in a way that favors the connections between the neurons that are most similar to each other. The mathematical representation of Random Walk is as follows:

$$w_i^+ = g(w_i + x_i\Theta(\gamma_i\Gamma)\Theta(\Gamma^A\Gamma^B))$$

Note

$$\Theta(a,b) = \begin{cases} 0 & a \neq b \\ 1 & \text{otherwise} \end{cases}$$

$g(x)$ is a function that maintains $w_i$ within the range $\{-L, ..., 0, ..., L\}$.

## 5. An Overview of Attack Strategies on Neural Network Key Exchange

### 5.1. Simple Attack

In a simple attack scenario, the attacker $\mathcal{E}$ trains an additional Tree Parity Machine using the same input vectors $x_i$ and observed output bits $\Gamma_{\mathcal{A}}$. These training samples are readily collected by eavesdropping on the message exchanged between the legitimate partners over the public communication channel. The neural network of $\mathcal{E}$'s adopts the same architecture as those of $\mathcal{A}$ and $\mathcal{B}$, and begins with random initial weights. At each stage, the attacker calculates her neural network's output. Next, $\mathcal{E}$ employs the same learning rule as $\mathcal{A}$ and $\mathcal{B}$, but during calculation, the attacker's $\Gamma_{\mathcal{E}}$ is substituted for $\Gamma_{\mathcal{A}}$. The learning rules are then as follows:

1. Hebbian learning rule:
$$w_i^{E+} = g(w_{i,j}^E + x_{i,j}\Gamma^A\Theta(\gamma_i^E\Gamma^A)\Theta(\Gamma^A\Gamma^B))$$
2. Anti-Hebbian learning rule:
$$w_i^{E+} = g(w_{i,j}^E - x_{i,j}\Gamma^A\Theta(\gamma_i^E\Gamma^A)\Theta(\Gamma^A\Gamma^B))$$
3. Random Walk:

$$w_i^{E+} = g(w_{i,j}^E + x_{i,j}\Theta(\gamma_i^E\Gamma^A)\Theta(\Gamma^A\Gamma^B))$$

Therefore, by utilizing her network's internal representation $(\gamma_1^E, \gamma_2^E, \ldots, \gamma_K^E)$, $\mathcal{E}$ can estimate $\mathcal{A}$'s output, despite any variations in the final output.

### 5.2. Geometric Attack

In contrast to the simple attack, the geometric attack exploits both the local fields of the adversary $\mathcal{E}$'s hidden units and the corresponding output $\Gamma^E$, thereby improving the effectiveness of the attack. The full configuration of the hidden neurons is characterized by their local field

$$h_i = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} w_{i,j}x_{i,j}$$

Similar to the simple attack, the adversary $\mathcal{E}$ attempts to imitate the behavior of $\mathcal{B}$ without any direct interaction with $\mathcal{A}$. When $\Gamma^A = \Gamma^E$, the adversary $\mathcal{E}$ can proceed by updating its weights using the same learning rule, followed by $\mathcal{A}$ and $\mathcal{B}$. However, if $\Gamma^E \neq \Gamma^A$, then $\mathcal{E}$ cannot prevent $\mathcal{A}$ from updating the weights. Instead, $\mathcal{E}$ seeks to adjust the internal representation of her Tree Parity Machine by exploiting the local fields $h_1^E, h_2^E, \ldots, h_K^E$. These quantities allow $\mathcal{E}$ to assess the confidence of each hidden unit's output, since a small absolute value $|h_i^E|$indicates a higher likelihood that $\gamma_i^A \neq \gamma_i^E$. Prior to the execution of the learning rule, $\mathcal{E}$ adjusts both the output $\gamma_i^E$ corresponding to the hidden unit having the minimum $|h_i^E|$ and the overall output $\Gamma^E$.

### 5.3. Majority Attack

In the majority attack, the adversary $\mathcal{E}$ improves the ability to infer the internal representations of party $\mathcal{A}$'s neural network by employing an ensemble of Tree Parity Machines instead of a single attacking network. At the outset, each attacking network is assigned independently generated random weight vectors, ensuring that there is no initial correlation among them and that their mean overlap remains zero. During the synchronization process, the attacker refrains from updating the weights at time steps where $\Gamma^A \neq \Gamma^B$, as these input vectors are discarded by the legitimate parties. However, when $\Gamma^A = \Gamma^B$, a weight update becomes required, and the attacker computes the output bits $\Gamma^{E,m}$ for each of her Tree Parity Machines. If the output bit $\Gamma^{E,m}$ of the $m$-th attacking network does not coincide with $\Gamma^A$, the attacker identifies the hidden unit $i$ that has the minimum absolute value of the local field $|h_i^{E,m}|$. Subsequently, the output bits $\gamma_i^{E,m}$ and $\Gamma^{E,m}$ are inverted in a similar manner as in the geometric attack. Afterward, the attacker collects the internal representations $(\gamma_1, \gamma_2, \ldots, \gamma_K))$ produced by all attacking Tree Parity Machines and selects the representation that appears most often. This majority-selected representation is then used by all attacking networks to perform the weight update according to the learning rule.

Once the attacking neural networks reach full synchronization, this strategy effectively becomes equivalent to a geometric attack. To prevent the Tree Parity Machines from becoming correlated, the majority attack and the geometric attack are applied alternately. At even time steps, learning is guided by the majority decision, while during odd time steps, $\mathcal{E}$ applies only the geometric correction. Consequently, the weight vector updates are not entirely identical, which lowers the degree of overlap between them.

### 5.4. Genetic Attack

An alternative strategy available to the attacker is the genetic attack, which does not aim to optimize the prediction of the internal representation but instead employs an evolutionary algorithm. $\mathcal{E}$ initiates the process with one Tree Parity Machine that is randomly initialized but has the flexibility to use up to $M$ neural networks. When $\Gamma^A = \Gamma^B$, the following genetic algorithm is used:
Mutation Step: While $\mathcal{E}$ has $M/2^{K-1}$ or fewer Tree Parity Machines, she identifies all $2^{K-1}$ internal representations $(\gamma_1^E, \gamma_2^E, \ldots, \gamma_K^E)$ that produce the output $\Gamma^A$.Subsequently, $\mathcal{E}$ uses these internal representations to adjust the weights of the attacking networks in accordance with the learning rule.
Selection Step: When $\mathcal{E}$ has greater than $M/2^{K-1}$ Tree Parity machine, then only the most robust Tree Parity Machines are retained. To accomplish this, $\mathcal{E}$ eliminates all networks that did not successfully predict at least $U$ outputs $\Gamma^A$ in the last $V$ learning steps, where $\Gamma^A = \Gamma^B$. By default, $U = 10$ and $V = 20$ are used as the threshold values. The success of the genetic attack largely depends on the selection mechanism used to identify the fittest neural networks. In an optimal setting, a Tree Parity Machine whose internal representations coincide with those of $\mathcal{A}$ would remain in the population throughout the process. Consequently, this would diminish the problem for $\mathcal{E}$ to synchronize $K$ perceptrons, and the genetic attack would inevitably be successful.

## 6. Multikey Neural Network Key Exchange Based on PPM

### 6.1. Proposed Design of Multikey Neural Network

We present the Multikey Neural Network design based on PPM in this part, which is called MT-PPM. Our design consists of three layers, namely the input layer, hidden layer, and output layer. There are $K \times N$ input neurons in the input layer and $K$ hidden neurons in the hidden layer. Each neuron can be viewed as a separate perceptron. There is only one output neuron in the output layer. The weight of MT-PPM is a $n$ tuple. It is described as follows:

$$w_{k,j} = (w_{k,j}^{(1)}, w_{k,j}^{(2)}, \ldots, w_{k,j}^{(n)}),$$

where $w_{k,j}^{(i)} \in \{1, 0\}$, the $k$-th hidden unit of the network is indicated by the index $k = 1, 2, ..., K$, while the vector elements are indicated by the index $j = 1, 2, ..., N$. The input vector of MT-PPM is also a $n$ tuple, which can be defined as follows:

$$x_{k,j} = (x_{k,j}^{(1)}, x_{k,j}^{(2)}, \ldots, x_{k,j}^{(n)}),$$

where $x_{k,j}^{(i)} \in \{1, 0\}$. Each hidden neuron's output value is determined by calculating the sum of all exclusive disjunctions (XOR) of input neurons and the weights:

$$\gamma_i = \left(\theta_N(\Sigma_{j=1}^N w_{i,j}^{(1)} \bigoplus x_{i,j}^{(1)}), \theta_N(\Sigma_{j=1}^N w_{i,j}^{(2)} \bigoplus x_{i,j}^{(2)}), \ldots, \theta_N(\Sigma_{j=1}^N w_{i,j}^{(n)} \bigoplus x_{i,j}^{(n)})\right)$$

$\theta_N(x)$ is a threshold function as described below:

$$\theta_N(x) = \begin{cases} 0 & x \leq N/2 \\ 1 & x > N/2 \end{cases}$$

Then the total neural network output is computed as,

$$\Gamma = \left( \bigoplus_{i=1}^{K} \gamma_i^{(1)}, \bigoplus_{i=1}^{K} \gamma_i^{(2)}, \ldots, \bigoplus_{i=1}^{K} \gamma_i^{(n)} \right),$$

and the neural network's final output can be succinctly expressed as $(\Gamma^{(1)}, \Gamma^{(2)}, \ldots, \Gamma^{(n)})$

### 6.2. Proposed Design of Key Exchange Based on MT-PPM

The proposed multikey neural network exchange is summarized below:

1. Two parties, say Dora and Smith have pre-trained neural network MT-PPM that can map input values to output values.
2. Random weight values are initialized for both Alice's and Bob's neural networks.
3. The following steps are executed until complete synchronization is attained:
   (a) Let $X$ be the input vector, which is generated randomly and common to both parties, Dora and Smith.
   (b) Calculate the values of Smith's and Dora's hidden neurons.
   (c) Then the values of the output neuron $\Gamma^A$ and $\Gamma^B$ are calculated for Dora and Smith respectively.
   (d) Following that, the values of Dora's and Smith's MT-PPM are shared and compared.
   (e) When the outputs coincide, the weight vectors are updated using an appropriate learning rule.
   (f) Or else if the outputs are not equal, then return to Step 1 and repeat the process until equal weights are achieved by both Dora and Smith.
4. Once complete synchronization is attained (i.e., the weights of both MT-PPM are the same), Dora and Smith can utilize their weights as keys.

## 7. Discussion

The two networks have the same weight when they are considered to be synchronized, and the synchronized weights can subsequently serve as secret keys for numerous cryptographic applications. Our proposed design is able to exchange $n$ group keys in a single synchronization session since the weight of the MT-PPM is a $n$ valued tuple. The MT-PPM can represent complex relationships between input variables that cannot be represented by binary neural networks.

By using complex-valued weights and activations, the model is able to capture both amplitude and phase information of the input signals. This better representation ability helps the network capture more useful information. Because of this, the network works well for tasks like signal processing and image recognition, where phase information is important. The MT-PPM exhibits better robustness to noisy inputs compared to binary and real-valued PPMs. The use of complex-valued weights and activations enables the model to suppress random noise components while preserving meaningful signal information, resulting in a more reliable representation of the input.

## 8. Conclusion and Future Work

In this work, we build a connection between cryptography and artificial neural networks. We propose a secret key exchange protocol with the help of different types of learning processes and achieved synchronization. We establish a post-quantum key agreement scheme based on learning rules and the dynamics of neural networks. The proposed scheme offers a post-quantum key exchange over an insecure public channel.

Several research directions remain open for future work. After this key agreement, a detailed comparison with existing schemes can be explored. In the future, several advanced attacks may occur; therefore, a comprehensive security analysis is required to preserve security. Anyone may integrate this key agreement with post-quantum cryptographic candidates such as isogeny-based and lattice-based cryptography. Additionally, extending the proposed scheme to multi-party key exchange and integrating it with post-quantum cryptographic frameworks constitute promising future directions.

## References


[1] Canetti, R., and Krawczyk, H. Analysis of key-exchange protocols and their use for building secure channels. In *Advances in Cryptology—EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20* (2001), Springer, 453–474.

[2] Saračević, M., Adamović, S., Maček, N., Elhoseny, M., and Sarhan, S. Cryptographic keys exchange model for smart city applications. *IET Intelligent Transport Systems 14*, 11 (2020), 1456–1464.

[3] Paar, C., and Pelzl, J. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[4] Stinson, D. R., and Paterson, M. *Cryptography: theory and practice*. CRC press, 2018.

[5] Diffie, W., and Hellman, M. E. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. Association for Computing Machinery New York NY United States, 2022, 365–390.

[6] Merkle, R. C. Secure communications over insecure channels. *Communications of the ACM 21*, 4 (1978), 294–299.

[7] Mehibel, N., and Hamadouche, M. A new approach of elliptic curve diffie-hellman key exchange. In *2017 5th International Conference on Electrical Engineering-Boumerdes (ICEE-B)* (2017), IEEE, 1–6.

[8] Ahirwal, R. R., and Ahke, M. Elliptic curve diffie-hellman key exchange algorithm for securing hypertext information on wide area network. *International Journal of Computer Science and Information Technologies 4*, 2 (2013), 363–368.

[9] MacKenzie, P., Patel, S., and Swaminathan, R. Password-authenticated key exchange based on rsa. In *Advances in Cryptology—ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings 6* (2000), Springer, 599–613.

[10] Gidney, C., and Ekerå, M. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum 5* (2021), 433.

[11] Liu, Y., and Pass, R. On one-way functions from np-complete problems. *Cryptology ePrint Archive* (2021).

[12] Montgomery, P. L. A survey of modern integer factorization algorithms. *CWI quarterly 7*, 4 (1994), 337–366.

[13] Shor, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science* (1994), Ieee, 124–134.

[14] Shor, P. W. Polynomial time algorithms for discrete logarithms and factoring on a quantum computer. In *Algorithmic Number Theory: First International Symposium, ANTS-I Ithaca, NY, USA, May 6–9, 1994 Proceedings 1* (1994), Springer, 289–289.

[15] Bernstein, D. J., and Lange, T. Post-quantum cryptography. *Nature 549*, 7671 (2017), 188–194.

[16] Godhavari, T., Alamelu, N., and Soundararajan, R. Cryptography using neural network. In *2005 Annual IEEE India Conference-Indicon* (2005), IEEE, 258–261.

[17] Kinzel, W., and Kanter, I. Neural cryptography. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.* (2002), vol. 3, IEEE, 1351–1354.

[18] Lawrence, J. *Introduction to neural networks.* California Scientific Software, 1993.

[19] Reyes, O., Kopitzke, I., and Zimmermann, K. Permutation parity machines for neural synchronization. *Journal of Physics A: Mathematical and Theoretical 42*, 19 (2009), 195002.

[20] Rosen-Zvi, M., Klein, E., Kanter, I., and Kinzel, W. Mutual learning in a tree parity machine and its application to cryptography. *Physical Review E 66*, 6 (2002), 066135.

[21] Ng, A. Cs229 lecture notes. *CS229 Lecture notes 1*, 1 (2000), 1–3.