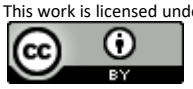
Available online at: https://ijact.in

Date of Submission	22/03/2019
Date of Acceptance	14/06/2019
Date of Publication	21/06/2019
Page numbers	3140-3145(6 Pages)

Cite This Paper: Smruti Chourasia, Hrishikesh B.C., Queenie D, Krati A, Lavanya K. (2019). Vectorized neural key exchange using tree parity machine, 8(5), COMPUSOFT, An International Journal of Advanced Computer Technology. ISSN: 2320-0790, PP. 3140-3145.



ISSN:2320-0790

COMPUSOFT
An International Journal of Advanced Computer Technology

VECTORIZED NEURAL KEY EXCHANGE USING TREE PARITY MACHINE

Smruti Chourasia[1], Hrishikesh Bharadwaj C[2], Queenie Das[3], Krati Agarwal[4], Lavanya K[5]

School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, India

[1]smruti.chourasia2016@vitstudent.ac.in, [2]hrishikeshbharadwaj.c2016@vitstudent.ac.in, [3]queenie.das2016@vitstudent.ac.in, [4]krati.agarwal2016@vitstudent.ac.in, [5]lavanya.k@vit.ac.in

Abstract: The communication boom in the past few decades has resulted in a large flow of data. This entails the need for having high security and privacy with regards to data confidentiality and authenticity. One method of doing so is by utilizing a synchronized key. In the domain of cryptography there exist several methods of key generation, one such method is the tree parity machine (TPM) involving neural cryptography. In our paper, we provide a novel vectorized TPM (vTPM) in order to develop a key. We have also provided a system to detect any unwanted listeners, as one of the weakness of the TPM algorithm is Man in the middle attacks. We have further utilized this key for authentication between a sender and a receiver. The authentication is carried out by means of H-MAC with the SHA-512 hashing mechanism. Finally, a comparison is drawn out between the serial and vector implementation of the Tree Parity Machine.

Keywords: Vectorized Tree Parity Machine (vTPM), Neural Cryptography, Hashing, H-MAC, Man-in-the-middle-attack, Parallelization

I. INTRODUCTION

Cryptography is the branch of computer science which deals with various methods of protecting data in order to maintain the four pillars of data protection. The four pillars of data protection include confidentiality, authentication, integrity and non-repudiation. Confidentiality aims to make sure that data access can be provided only to a trusted user. Authentication on the other hand looks into the identity of the sender and receiver in our system. Integrity provides a guarantee to the sender- receiver pair that the data has not been tampered or altered in any way. Non-repudiation helps to identify the sender even if the sender denies it.

HMAC either known as Hash message authentication code or Hash-based message authentication code is one of the well-established mechanisms for asserting the two major cryptography principles of integrity and authenticity. This is can be accomplished by using a MAC that needs a hash function “H” and a secret key “K”. The major difference between the other types of authentication and HMAC is it signs the entire request.

The concept of Neural Networks draws inspiration from the human nervous system. The neural networks consist of different layers that are analogous to the neurons of the human system. The first layer takes the input provided and transmits it to the succeeding layers in a manner similar to how neurons communicate with each other using synapses. The number of layers in a system determines its complexity. Each "synapse" has a weight as a parameter that is included in the calculation of the input to the succeeding layer. The

network has a learning process which aims to optimize the outputs by updating of weights for each layer. This method of learning is termed as a gradient descent mechanism. In order to simulate real world scenarios and not merely restrict to linear bounds of data a neural network employs the use of a concept called activation function at the end of each layer. The activation function helps to restrict the values of output from a particular neuron into a bounded range so as to maintain a coherent nature with a realistic scenario.

The process of encryption and cryptanalysis that uses stochastic algorithms in combination with neural networks gives rise to a branch of cryptography called Neural Cryptography. The Neural key exchange protocol is an important part of this domain, and is a protocol that allows the secure transfer of a shared key between the two parties Alice and Bob. The basis for this lies in the usage and synchronization of two Tree parity machine (TPM).

Since the Tree parity machine is a computationally intensive algorithm, there arises a need to optimize and reduce the computation time. In order to make the algorithm more feasible we have utilized the concept of vector processing or in short vectorisation. Vectorisation is a parallel processing technique which uses multiple threads simultaneously on a Single Instruction Multiple Data (SIMD) based processor to process chunks of data with the same instruction. The Tree Parity Machine can be undermined by Man-in-the-middle attacks due to the fact that there can be any external agent listening in on the transmission of output values.

In the following sections, we first encounter the mechanism employed for neural key exchange. Following which we present the vectorized implementation of this algorithm. We also look into the Man-in-the middle-attacks that the Tree Parity Machine is susceptible to. In addition, to the above we also look into the utilization of the key generated by the TPM for authentication using H-MAC. Finally, we discuss the results of our vectorized implementation by contrasting it against the serial version of the Tree Parity Machine.

II. LITERATURE REVIEW

Originally, the study and research of neural networks was piloted by its potential of being a powerful memory machine that can learn [1]. The basis of Neural Cryptography is the notion that two neural networks can synchronise by mutual learning.

In 2002, Kinzel et al. [2] proposed a secret key exchange protocol as opposed to a public key exchange channel that was proposed by Diffie and Hellman [3] in 1976. The key exchange method consists of the two shallow neural nets. Each party that is communicating has a neural network, initially having random weights. The neural networks update themselves at each round. The moment the two networks synchronize we can infer the key from their mutual output bits. Once synchronized, the common identical weights gained from the two parties are used as the encryption key. In [2], it is also shown that an attacking party using an identical neural network having the same learning procedure has very low chances of synchronizing their neural network with that of another party. For key generation over a public channel, this protocol was the first one which was based on number theory.

However, in 2002 itself, Klimov et al. [9] broke this system by attacking it with three different techniques- probabilistic analysis, genetic algorithms, and geometric considerations.

Klein et al. [4] in 2004 presented the usage of neural nets in key exchange systems in public channels. However, in contrast to the previous efforts Klien et al. used a chaotic synchronisation system. The alternate output is used to train the network and are synchronised through a chaos synchronised system resulting in equal time dependent weight vectors. This type of synchronization made the system more secure.

The use of neural cryptography for the generation of secret key was suggested by N. Prabakaran [7] based on synchronisation of Tree Parity Machines (TPM). This system has two identical systems that are dynamic and start from different initial conditions. These identical systems are synchronised using common input values resulting in providing a common vector as the input to the network. After the calculation of their outputs, their weight vectors are updated when a similar mutual output is observed at every step. The initial random values of the weight vectors are generated by PRNGs (Pseudo-Random Number Generators). No exchange of input or output vectors occurs through the public channel until the weight vectors of the two networks are matched and this matched vector is used for the process of encryption and decryption as the secret key.

In [5], Kinzel et al. describe a Tree Parity Machine (TPM) as a novel type of shallow neural net with forward feed having input (N) neurons, hidden (K) neurons, and a range (L) of weights. There are two neural networks, one belonging to each party. These networks receive a common input that has been randomly generated. In each step, they learn the outputs that are common to both of them. This results in the concept of synchronisation by mutual learning as mentioned in [6]. The two neural machines are synchronised to the same weight vector which is time dependent. This concept of synchronisation is used for generation of a secure secret key in [4] and [8].

Our proposed system implements an authentication system using HMAC and SHA-512 hashing mechanism.

As presented in [10], HMAC is a mechanism for message authentication that uses cryptographic hashing functions. This mechanism can be used with any iterative hashing function like MD-5 and SHA-1. The performance of the HMAC is essentially dependent on the underlying hashing function.

The NIST launched the Hash Standard [11] in 2002, which detailed SHA-512, SHA-384 and SHA-256. NSIT [11] describes the full description of these Secure Hashing Algorithms. SHA-512 produces a 512-bit message hash, while SHA-256 gives a 256 bit hash message.

III. METHODOLGY

A. Tree Parity Machine

It is a key exchange algorithm analogous to the Diffie-Hellman protocol. TPM relies on a neural network with a single hidden layer. The tree comprises of input (N) neurons and hidden (K) neurons. This results the TPM to have K*N number of weights from the neural network. We restrict the weights between the bounds {-L, …, -2, -1, 0, 1, 2, …, L}, where L is a parameter of the TPM.

Suppose that there are two machines Alice and Bob, which require the generation of a common key. The two machines Alice and Bob are initialized with random weights and provided the same parameters (K, N, L). Initially, the weights are different due to the random initialization. In order to exchange the key between Alice and Bob, we require the updation of weights in such a manner so as to synchronize the two machines, thereby, having the same weights for Alice and Bob.

The output function Tau (τ) is calculated as follows:

$$Input\ Vector\ x_{i,j} = \{-1,0,1\} \quad (1)$$

$$Weights\ w_{i,j} = \{-L, \dots, 0, \dots, L\} \quad (2)$$

$$\sigma_i = sign\left(\sum_{j=1}^{N} x_{i,j} \times w_{i,j}\right) \quad (3)$$

Where,

$$sign(x) = \begin{cases} -1 & : x < 0 \\ 0 & : x = 0 \\ 1 & : x > 0 \end{cases} \quad (4)$$

$$\tau = \prod_{i=1}^{K} \sigma_i \quad (5)$$

We use the following three mentioned algorithms to update the weights in the TPM, each of them varying slightly to the other.

- Hebbian

$$w_i^+ = w_i + \sigma_i x_i \theta(\sigma_i \tau)\theta(\tau^A \tau^B) \quad (6)$$

- Anti-Hebbian

$$w_i^+ = w_i - \sigma_i x_i \theta(\sigma_i \tau)\theta(\tau^A \tau^B) \quad (7)$$

- Random walk

$$w_i^+ = w_i + x_i \theta(\sigma_i \tau)\theta(\tau^A \tau^B) \quad (8)$$

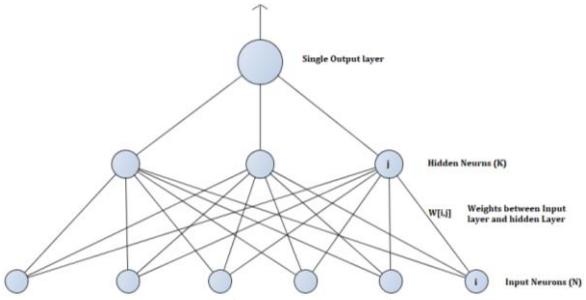


Fig. 1: Architecture of Neural Network for Tree Parity Machine

Algorithm to Synchronize:

Step 1: Initialize values K, N, L

Step 2: Initialize a random Weight Matrix of Dimension K*N for Alice and Bob

Step 3: Get a Common Input vector for Alice and Bob X

Step 4: Calculate the Output Function Tau for Alice and Bob:

Step 5: If Tau for Alice equals Tau for Bob:

Update Weights based on Update rule

Else:

Go back to Step 3

Step 6: If Weight matrix for Alice equals Weight matrix for Bob:

Generate Key

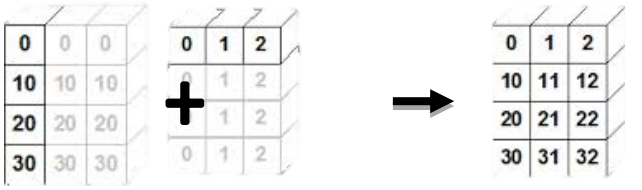
Else:

Go back to Step 3

B. Vectorization of TPM

Simple TPM uses for loops for updating the values in the rules, this serialised implementation starts lagging once there is an increase in the size of the neural network. In real life scenario we need large authentication key to ensure high security and hence we vectorized the serial implementation using the "NumPy" library [12].

NumPy arrays are densely packed arrays of homogeneous type. Numerous reasons motivated us to implement this, first and foremost the process of vectorization, a powerful ability within NumPy helped to express operations as occurring on entire arrays rather than their individual elements. In this method, for loops was replaced with array expressions. Secondly, use of Broadcasting feature of NumPy that helped us compute over two arrays of different shapes. Lastly, NumPy used very less memory to store data [12].



Matrix A(4,1) Matrix B(1.3) Matrix C (4,3)

Fig 2: Sample of Broadcasting in Numpy

C. *Countering Man-in-the-Middle*

The eavesdropper (Eve) has also been incorporated in this implementation. Knowing the initial vector, it is almost impossible to synchronise the exact same weights. But if the man in the middle is able to do so, we check by utilising the synchronisation values. In our implementation we have introduced a third machine Eve to check for any interception. For each round of update between Alice and Bob we compare the Output value τ of Alice and Bob with the output value τ of Eve. In the case there is a match we update the weights for Eve using the same method as used for Alice and Bob. Finally, we check the weights between the Two synchronised Machines Alice and Bob and our third external listener Eve. In the case that there is 100% synchronisation between all the machines, we can conclude that Eve knows the secret key between Alice and Bob. We terminate here and exit to prevent intrusion.

D. *Key Generation from Weights*

Algorithm to generate key:

Step 1: Initialize string object (str) to all possible characters key can have,
[A-Z,0-9]
Step 2: keySize = length(str) / L
Step 3:keyLength = K * N / keySize
Step 4: Loop variable i from range 1 to keyLength
Repeat,
temp =1
from range (i-1*keySize) to (i*keySize-1)
temp = temp + W[j] + L
Append str[temp] to Key
Step 5: Return Key

E. *H-MAC from Generated Key*

The Hash based Message Authentication Code (H-MAC) is then used for authentication purpose using the key generated. The message is padded to the left with an input signature. This entire message along with the padding forms the input for a hash function.

Take M to be the original message and H to be the hashing function. L is the number of blocks in M, K is the secret key that will be used in hashing, IV is a constant value forming the initial vector and Yi is the i^{th} block number of the original message M, where the range of i falls from [1, L].

The following is the generation of Si, the input signature and So, the output signature.

$$S_i = K^+(xor)\ ipad \quad (9)$$
$$S_o = K^+(xor)\ opad \quad (10)$$

where, $K+$ = zeros padded to K from the left resulting in a length of b bits,
$opad$=01011100,
$ipad$=00110110
Each taken b/8 times repeatedly.

$$MAC = H(S_o\ ||h(S_i\ ||M)) \quad (11)$$

IV. RRESULTS AND DISCUSSION

A. *Comparison of Serial and Parallel Implementation*

In our study of neural cryptography for authentication, we have analysed and compared the serial and parallel implementations of the Tree Parity Machine. We have implemented the algorithm in Python on the Google Colab open source platform. The processor which we used was 2 Intel Xeon Processors running at 2.30GHz and having 12 cores each. When ran on the same initial parameters K, N and L we obtained a significant speed up. The minimum speed up we obtained was 243 times for k = 5 and n = 5, and the maximum speed up obtained was around 50,000 times for K = 49 and N = 49.

We ran the comparative time calculation for K, N in the range of 5 to 50 each. We found that the parallel vectorized implementation provided us with an almost constant output time of about 0.002s. On the other hand, our serial implementation depended heavily on the parameters K and N with a max time going above 12s.

Sr. No.	*K*	*N*	*Serial Time (in s)*	*Parallel Time (in s)*
1	5	5	0.03738	0.00015
2	10	10	0.22837	0.00015
3	25	25	1.16199	0.00017
4	45	45	8.85800	0.00024

Table 1: Values for Serial and Parallel Execution and corresponding K, N

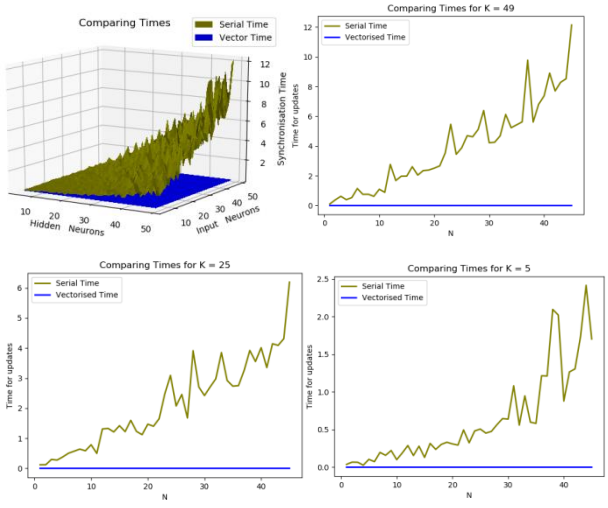


Fig. 3: Plot of the time taken by serial and vectorised implementation for various values of K, N

Figure 3 provides us a comparison of the time taken for the varying values of K and N. As we can evidently see from the graphs, the vectorized implementation provides a significantly larger time advantage and takes almost negligible time to execute and generate the key. We can also observe that there is not a linear increase in the serial execution time. An approximation would show that there is an exponential increase in the execution time, one may attribute this to the exponentially increasing number of linkages when we increase the values of K and N. However, when we observe the Vector time of execution, we can observe that the changes are barely noticeable and be in the order of 0.01 to 0.1 seconds that dwarfs when compared to the drastic change in serial execution time. This gives us the belief that vector execution time remains constant, but the following section delves into the variation of execution time for the vectorized implementation.

B. Comparing the execution time against the variation of parameters N, K and L

In a bid to see how the parameters influence the working of the vectorized implementation, we have compared the exaction times for synchronization of the tree parity machine by varying each of the parameters N, K and L independently.

When we varied the parameter K we kept the values of L and N fixed as we calculated the execution time for a range of K = (5 to 50). The value of L was fixed as 5 for convenience and N was taken for 4 different values of 10, 20, 30 and 40. From Fig 4. We can see that there is a general increase in the execution time as the number of Hidden Neurons K increases. This increase in execution time is not a very significantly large increase as the increase is in the order of 0.1 seconds.

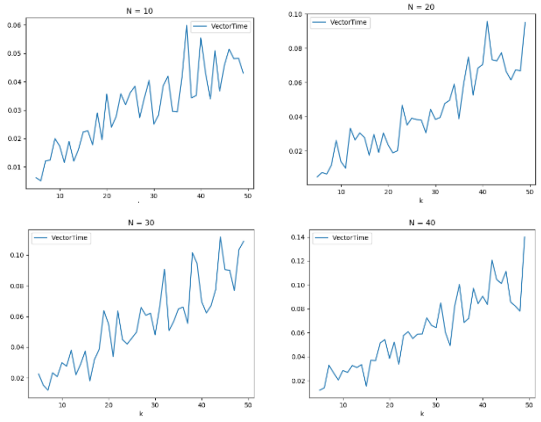


Fig.4: Plot of the vectorized time taken versus K for a fixed L = 5 and N = 10,20,30 and 40 respectively.

Next, the variation of N keeping K and L constant gave us results that were different. No apparent increase in the values could be found leading one to believe that the variation of N doesn't affect the vectorized execution time. This can be seen from Fig. 5 where the values randomly oscillate within a small bounded value. Here again we have fixed L as 5 for convenience and taken four cases of K.

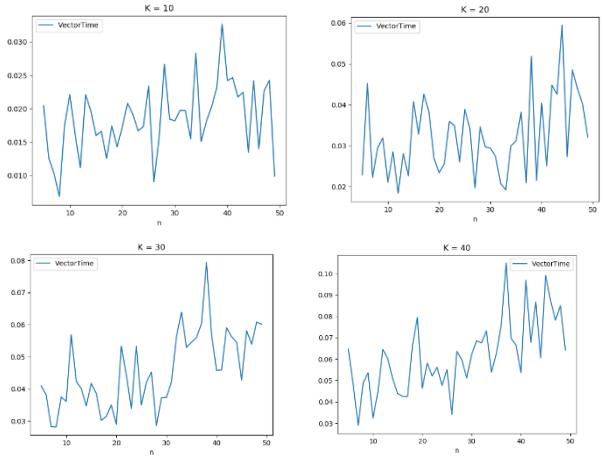


Fig.5: Plot of the time taken versus N for a fixed L = 5 and K =10, 20, 30 and 40 respectively

Finally, we look at the effect that L has on the execution time. We take a single case of N = 15 and K = 8 and vary the values of L over an interval of 5 to 50. We can clearly observe from Fig 6. that there is a randomness in the execution times and the variation is very minimum (order of 0.01 seconds). Since the variation has no definitive increasing pattern and the order of change is also negligibly small we can safely conclude that L bears no effect on the execution time in the vectorized implementation.

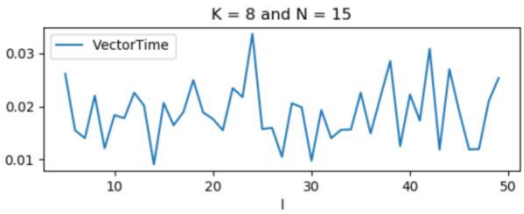


Fig.6: Plot of the time taken versus L for N = 15 and K = 8.

Thus, from our observations we can conclude that the maximum range of the weights (L) and the number of input neurons (N) bear no significant effect on the variation of execution time in the vectorized implementation. We can also safely conclude that the number of hidden Neurons (K) has a fairly significant effect on the execution time. We can see a gradual increase in the execution time of the vectorized implementation with an increase in the value of K.

C. Man-in-the-Middle Attack

We can observe that the man in the middle attacks can be effectively detected as the implementation exits as soon as an intrusion is detected. This is achieved by simultaneously

running a third Tree parity Machine 'Eve' alongside 'Alice' and 'Bob.' We detect an intrusion if and only if the weights of 'Eve' match those of either 'Alice' or 'Bob' during any of the cycles of synchronization. Once a match occurs, we can say that 'Eve' which emulates an external Man in the middle has hijacked the synchronization. Thus, terminating the whole key exchange process and starting afresh. This method may not be most optimal as it may result in a precarious scenario where the key is never exchanged, but provides a failproof method to prevent the synchronization of a third party. There by, providing a road block to all Man-in-the-middle attacks.

D. Authentication using HMAC

The key generated is different each time the TPM is run and is known only to Alice and Bob. Using vTPM, we can generate really long keys of several characters in a short duration of time, thus, providing a larger security against brute force attacks such as the Birthday Attack. The Hash Code generated is 512 bits as predefined by the SHA-512 hash function used.
Given below is an example of the Key and corresponding Hash Code after HMAC:

Output Key from vTPM is **ILOTLLFPVQLJHBBLM**

Message to Encrypt: **This is a secure message**

Generated Hach Code:**82cd30996750cbdf3d473fa9b1277e8301ed0ab0 a7f8be48d4ae06129b7a8fff169cdca09709b14aa0ce3c ba354782fe02a803fa22fe1e505752ffb3f7b0e95a 137**

The advantage of using vTPM as a key generator is the large size of keys that can be generated in a short amount of time.

V. CONCLUSION

In our paper, we have successfully implemented a vectorized version of the Neural Cryptography method of Tree Parity Machine (vTPM). It was also shown that the processing time decreases drastically even for large computations when using vTPM. This can be utilised in communication systems which need high security. Since the length and strength of the key is variable, it is highly adaptive and we can optimise the security for a given need. Future prospects of this project include prevention of Man-in-the-middle attacks as we have already detected and avoided them. Another aspect that can be improved on is the key generation method as it relies on the hashing of a predefined set of characters.

VI. REFERENCES

[1] Rosen-Zvi, Michal, Ido Kanter, and Wolfgang Kinzel. "Cryptography based on neural networks—analytical results." Journal of Physics A: Mathematical and General 35.47 (2002): L707.

[2] Kanter, Ido, Wolfgang Kinzel, and Eran Kanter. "Secure exchange of information by synchronization of neural networks." EPL (Europhysics Letters) 57.1 (2002): 141.

[3] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." IEEE transactions on Information Theory 22.6 (1976): 644-654.

[4] Einat Klein, Rachel Mislovaty, Ido Kanter, Andreas Ruttor, and Wolfgang Kinzel. 2004. "Synchronization of neural networks by mutual learning and its application to cryptography." In *Proceedings of the 17th International Conference on Neural Information Processing Systems* (NIPS'04), L. K. Saul, Y. Weiss, and L. Bottou (Eds.). MIT Press, Cambridge, MA, USA, 689-696.

[5] Kinzel, Wolfgang, and Ido Kanter. "Neural cryptography." Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.. Vol. 3. IEEE, 2002.

[6] Kinzel, Wolfgang, and Ido Kanter. "Interacting neural networks and cryptography." Advances in Solid State Physics. Springer, Berlin, Heidelberg, 2002. 383-391.

[7] Prabakaran, N., and P. Vivekanandan. "A new security on neural cryptography with queries." International Journal of Advanced Networking and Applications 437, Volume: 02, Issue: 01, Pages: 437-444 (2010)

[8] Jogdand, R. M., and Sahana S. Bisalapur. "Design of an efficient neural key generation." International Journal of Artificial Intelligence & Applications (IJAIA) 2.1 (2011): 60-69.

[9] Klimov, Alexander, Anton Mityagin, and Adi Shamir. "Analysis of neural cryptography." International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2002.

[10] Bellare, Mihir, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication." Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 1996.

[11] NIST. "Secure Hash Standard," PIPS PUB 180-2, 2002.

[12] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation." Computing in Science & Engineering 13.2 (2011): 22.