Date of publication DEC-31, 2025, date of current version OCT-29, 2025.
www.computingonline.net / computing@computingonline.net

Print ISSN 1727-6209
Online ISSN 2312-5381
DOI 10.47839/ijc.24.4.4347

Neural Networks for White Box Cryptography in Software Protection Systems

VIKTOR ROVINSKYI[1], OLGA YEVCHUK[2], YURI STRILECKYI[3]

[1] Faculty of Mathematics and Computer Science, Vasyl Stefanyk Precarpathian National University, Ivano-Frankivsk, 76018 UA
[2] Softjourn-Ukraine LLC, Ivano-Frankivsk, 76018 UA
[3] Institute of Information Technologies, Ivano-Frankivsk National Technical University of Oil and Gas, Ivano-Frankivsk,76019 UA

Corresponding author: Viktor Rovinskyi (e-mail: victor.rovinsky@pnu.edu.ua).

ABSTRACT This research explores the integration of neural networks into software protection mechanisms, focusing on enhancing cryptographic robustness against unlicensed copying and unauthorized access. The primary purpose is to develop a robust method that combines obfuscation and encryption to protect software by binding its activation and operation to specific hardware and user data, thereby preventing unlicensed replication and disassembly. The approach involves generating a unique hash of hardware identifiers, training a neural network on a remote server to produce a bytecode sequence for a virtual machine, and using the network's weights as an activation code. Key results show that optimal neural network configurations, particularly those with two dense layers and one LSTM layer, achieve 100% accuracy in mapping predefined inputs to specific bytecodes within an average training time of 11 seconds, while generating pseudorandom outputs for all other inputs. Statistical analysis of output distributions reveals high entropy, demonstrating resilience against statistical attacks. The proposed method offers a layered defense against common attack vectors and ensures persistent security throughout the software's lifecycle.

KEYWORDS software protection; neural network; cryptography; obfuscation; virtual machine.

I. INTRODUCTION

The protection of software from unauthorized use involves countering actions by malicious actors, such as using the full functionality of the software without acquiring the appropriate license, sharing a legally purchased copy with other users, using more copies than allowed by the license, as well as unauthorized analysis of the software's copyrighted data and algorithms. Typically, protection tasks can be implemented through hardware keys, software methods, or a combination of both.

An external hardware key, typically connected via USB, can either perform user authentication (e.g., using a private key stored in the device and inaccessible externally) or execute part of the main software's functions, such as implementing algorithms that are the intellectual property of the software developer and must be protected from disassembly. Alternatively, the target executable code may be stored in the software file in encrypted form and decrypted by the hardware key during execution, then passed on to the main program for execution. A key advantage of using hardware keys is their increased resistance to hacking, as compromising hardware tools requires expensive specialized equipment that is inaccessible to most attackers. However, a hardware key reduces the number of available USB ports, may load the USB bus bandwidth, and its use may not always be convenient for users.

Software methods involve verifying the authenticity of the software copy during installation and/or while the program is running, which can occur locally or on a remote server. Periodic sending of identification data of the installed copy to a remote server via the Internet during regular program operation allows for the timely detection of unauthorized software use, with all necessary checks being performed on the server. The server can also be used for remote execution of critical code fragments that should not be available for analysis. However, such a scheme requires a constant connection, which may be inconvenient or impossible. The advantage of server-based protection is that compromising it requires breaching the server software itself, which is usually not an easy task.

Moreover, any software method always involves certain actions on the local computer, including entering and initially processing the serial number, as well as performing various operations depending on the results of local or remote checks. To control the number of installed copies, software is often

linked to a specific hardware configuration of the computer (video adapter, HDD, BIOS, network card, etc.). In this case, identification data typically consists of a set of unique identification numbers of the corresponding hardware or the result of their transformation using a cryptographic function (e.g., encryption, hashing). The identification data are usually sent to the server during the installation process, and the server's response may contain an activation code, which can be used both for completing the installation successfully and for subsequent software operation (either at program startup or periodically during its operation).

The success of an attacker's efforts to bypass a protection algorithm, which is partially or fully executed locally, largely depends on the ability to disassemble the program. Protection against disassembly can be achieved by applying a cryptographic transform to the program code [1], where a critical section of the program is decrypted just before execution, then executed, and then either encrypted again or erased, thereby preventing static disassembly. However, this does not eliminate the possibility of dynamic analysis, where the attacker pauses the program and dumps the process memory immediately after decryption to obtain the decrypted code. The reliability of this method also significantly depends on the ability to hide the cryptographic key from the attacker, which can be difficult if they have full access to the system. To address this vulnerability, white-box cryptography has been proposed, where the key for the AES or DES algorithm is hidden in a modified execution scheme [2].

A method to mitigate the analysis of already disassembled code is obfuscation, which involves transforming the program code in a way that does not alter its functionality but makes it more difficult to analyze. For instance, unnecessary operations that do not perform any useful work can be added to the program to artificially increase its execution time, inflate its size, and divert the attacker's attention, thereby making it harder to understand the disassembled program. Variants of obfuscation include various control flow transformations, data obfuscation (e.g., securely hiding cryptographic keys or message strings displayed to the user upon triggering protection), and preventive transformations that complicate automatic deobfuscation [1].

An additional obfuscation method is virtualization, where the protected code is converted into a sequence of bytecode for a virtual machine, which is then executed by a specialized interpreter [3, 4]. In this case, static code analysis is impossible if the attacker lacks knowledge of the virtual machine's command system, and dynamic analysis becomes significantly more challenging because the attacker is forced to navigate through the cycle of parsing and dispatching the virtual machine instructions during step-by-step execution. The degree of obfuscation can be increased by applying multiple layers [5].

Thus, the combination of obfuscation and encryption allows for the creation of an optimal protection algorithm. In this study, the use of artificial neural networks is considered for this purpose, as they enable implementing arbitrarily complex mappings between input and output data, and the actual algorithm implemented by such a network is difficult to analyze, even if the network structure and its weights are known.

The ideas of using neural networks in encryption and obfuscation tasks have been repeatedly proposed over the past few decades [6, 7]. Some of the earliest works in this area include [8] and [9], which proposed using a Hopfield neural network for a symmetric encryption scheme. In [8], a Hopfield neural network with a limited set of weights (each weight can only have values of 1, 0, or -1) was used as part of the encryption and decryption algorithm. The scheme employs two secret keys, which, after being processed by the neural network and subjected to a projection function, are transformed into a bitstream that encrypts or decrypts data using the XOR operation. Here, the neural network is effectively used to transform the secret key such that different encrypted messages are generated for the same plaintext depending on the chosen permutation matrix. In [9], the same property of the Hopfield network, that is, convergence to one of the stable states (attractor) within a finite number of steps, was utilized, but the transformations were performed not on the key, but on the message to be encrypted. The encryption result is a random sequence that causes the network state to converge to an attractor, equivalent to the transformed plaintext. The cryptographic strength of such a system is analyzed in [10].

The fact that a neural network can act as associative memory, producing specific sequences in response to partial, distorted, or even random input, has been utilized in several newer works on neural networks with different structures, particularly autoencoders [11, 12]. As it is shown in [13], training examples are stored as attractors in overparameterized autoencoders, while overparameterized sequence encoders store training examples as stable limit cycles and are more efficient in memorizing than autoencoders. Other architectures were also researched for associative memory tasks, such as spiking neural network [14] or transformers [15].

In [16], a scheme for generating a secret key through training identical neural networks (so-called tree parity machine) on both the sender's and receiver's sides is described. Synchronization is achieved by using identical random input data for training the neural network. The trained network's weights are used as the key for symmetric encryption/decryption via the XOR method or algorithms like AES or DES. A similar scheme is also investigated in [17], and generalized to vector tree-parity machine in [18]. Also, using Generative Adversarial Networks (GANs) for synchronizing neural networks on the sender's and receiver's sides was proposed. Specifically, in [19], a symmetric encryption scheme is studied where both parties know a secret key unknown to the adversary and train neural networks of identical structures (one fully connected layer and several convolutional layers) to minimize the receiver's message reconstruction error while simultaneously minimizing the mutual information between the plaintext and the adversary's reconstruction, which is also modeled by a neural network of similar structure. Application of neural networks to asymmetric cryptographic schemes was proposed recently as well [20, 21].

In [22], multilayer neural networks were used for encryption/decryption through character sequence substitution. The secret key consists of the network structure and its set of weights. Separate networks are trained for encryption and decryption, each transforming an input bit sequence into an output sequence, so the mapping table must be pre-agreed upon by the message sender and receiver. The article provides an example with 6-bit input and output sequences, and while the lengths can be arbitrary in general, the required computational resources for longer lengths are not evaluated.

In [23] and [24], using a multilayer neural network for encrypting and decrypting data was proposed as follows: the hidden layer output is used as the encryption result, and the

output layer result is used for decryption. The neural network is trained on a dataset where the input and output data are identical. This allows the implementation of a substitution cipher for relatively short sequences (an example for 8-bit sequences is considered in [23]), since the size of the training set grows exponentially with increasing sequence length, and the probability of successful training likely decreases.

In [25], a neural network is used only for decryption, while encryption is performed through a sequence of permutations and logical functions. The neural network is trained on a set of all possible pairs of plaintexts and encrypted messages. The total number of such pairs is limited by the characteristics of the secret key, which comprises a permutation table and a sequence of logical functions. Thus, the structure and weights of the neural network serve as a public key in an asymmetric encryption scheme.

The use of a neural network solely for decryption is also considered in [26], where encryption of 8-bit input blocks is proposed using logical functions and bitwise shifts, followed by the addition of 4 random bits at random positions. A 3-layer neural network is used for decryption, although the method for forming the training dataset is not specified.

In [27], the back-propagation neural network is used instead of traditional password/verification tables for user authentication, associating usernames with hashed passwords through trained network weights. Each input and each output represent a binary value, and both usernames and passwords are limited to 8 characters only. At the time of publication (2005), such a network required a relatively long training time. However, it can be assumed that this is due not only to the lower available computing power, but also to the need to memorize a significant amount of pseudo-random data with equally high accuracy.

II. MATERIAL AND METHODS

A. DESCRIPTION OF THE PROTECTION SYSTEM

A typical sequence for installing software provided under a paid license, with control over the number of installed copies tied to the computer's hardware using a remote server, follows this process:

1. The user launches the installer program and enters the purchased license serial number and user information (such as user name, organization name, email address, etc.).
2. The installer collects information about the hardware characteristics and/or unique hardware identifiers, generates a hardware identifier based on this information, and sends it to the server along with the user data and serial number.
3. The server verifies the submitted data and responds with either an activation code that allows the installation to continue or an error message (in case the allowed number of installed copies is exceeded).
4. The received activation code is stored on the local computer.
5. Each time the program is launched, the activation is verified by executing a specific algorithm on the activation code and the current hardware characteristics. These checks may also occur periodically or when using certain software functions during regular operation.

This process is illustrated graphically in Fig. 1. From the security standpoint, the most vulnerable steps in this process are the operations performed on the activation code on the local computer, both during the installation of the software and during its operation. During installation, the time constraints on operations can be more lenient than during program startup. For instance, after sending a request to the server, the activation code can be sent to the client via email a few minutes after receiving the request, whereas at program startup, it is desirable that the execution time does not exceed a few seconds.

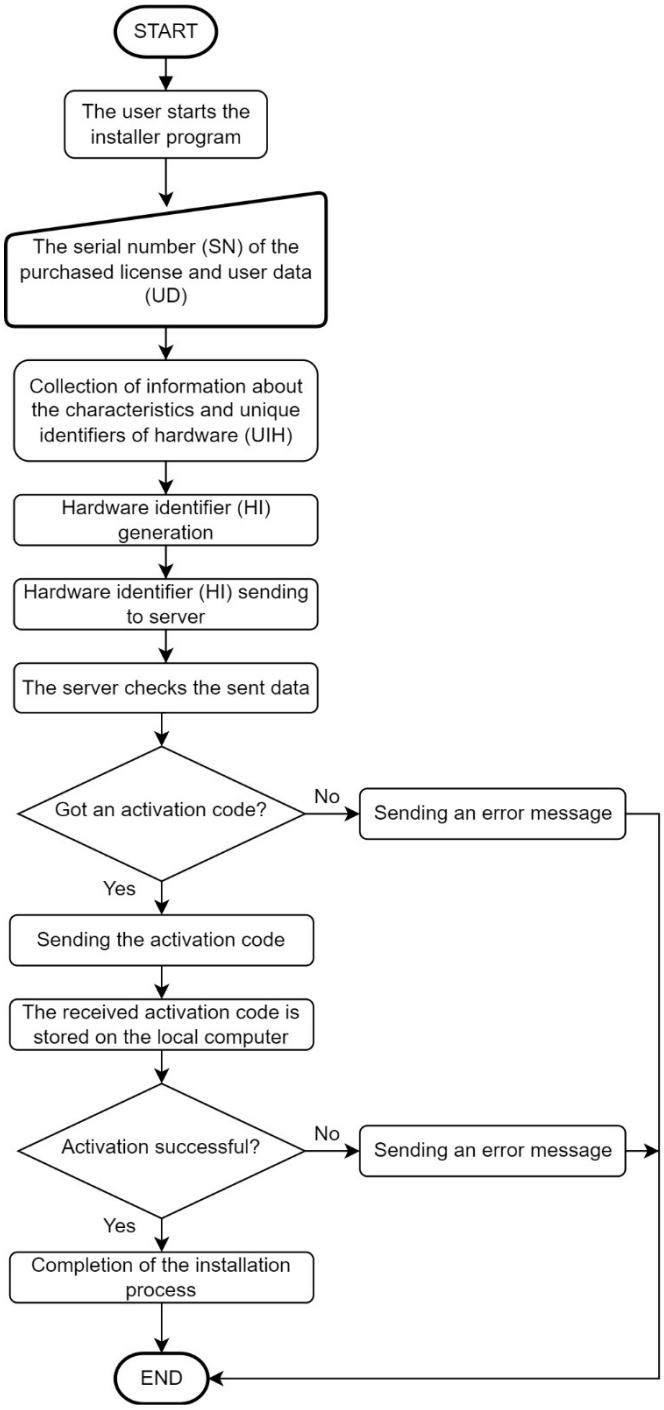

Figure 1. Protected software installation flow

Additionally, it is important to ensure that the activation code cannot be reused on another computer and that it cannot be forged by analyzing an arbitrary pair of "user data - activation code" and the corresponding program code that uses the activation code to unlock installer or program features.

The following scheme, which combines obfuscation and encryption techniques, meets these requirements:

1. On the local computer, an identifier string is generated, consisting of the user's name and unique hardware identifiers

(e.g., hard drive serial number and MAC address of the network adapter).

2. A hash of the identifier string is calculated.

3. On the remote computer, a neural network is trained, which, when provided with the hash from step 2 as input, generates a sequence of bytecodes for a virtual machine as output, and produces pseudorandom output data for any other input.

4. The neural network's coefficients and its parameters (number of neurons, number of layers, type of activation function) form the activation code, which is sent to the client.

5. After receiving the activation code, the installer program initializes the corresponding neural network, feeds the hash from step 2 into it, and processes the resulting bytecodes with the virtual machine's subroutine. The result of executing the code in the virtual machine is the successful completion of the installation process.

6. To verify the activation of the program during each launch and/or during operation, steps 3-4 must be repeated when preparing the activation code for all program fragments subject to protection. Each protected fragment will be executed as described in step 5.

The step of calculating the hash is necessary to normalize the length of the input data and to obscure which specific hardware parameters are used for identification: if raw data were transmitted, an attacker could extract them by analyzing the contents of the request sent to the server. To conceal the hash calculation step, instead of using standard hash functions, a single-layer neural network with random coefficients generated during installation and stored along with the activation code can be used. In this scenario, the procedure for obtaining the bytecode during activation verification is reduced to processing the identifier string with a neural network composed of the hash calculation layer and decryption layers (see Fig. 2).

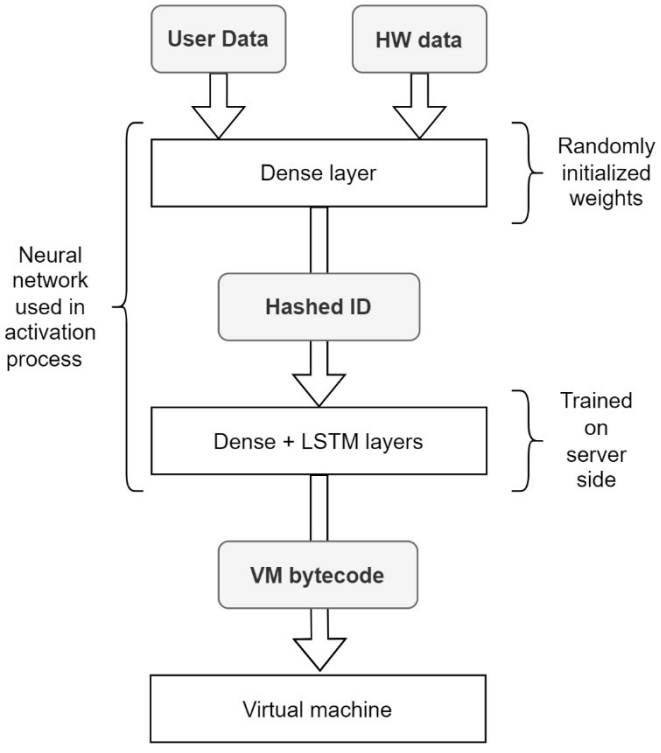

Figure 2. Activation flow

The described method of using a neural network is somewhat similar to those previously discussed in the literature for encryption/decryption tasks [8-10, 27]. In fact, this scheme can be viewed as a specialized cryptographic system where the encryption process involves training the neural network, and decryption is the processing of the input sequence by the neural network. However, the significant difference here is that, for the purpose of protection, it is not necessary to implement the ability to encrypt arbitrary messages, but only the specific bytecode sequence subject to obfuscation needs to be encrypted, with a guarantee that no other input sequence will produce the same encryption result. Therefore, there is no need to develop a special algorithm for converting plaintext into ciphertext: the encrypted message can be any arbitrary byte sequence, and the neural network training process will ensure that this sequence will be decrypted into the specified plaintext. Thus, unlike classical cryptographic methods, where a secret key is typically generated using a random number generator and the ciphertext is formed by performing a sequence of actions (encryption algorithm) on the plaintext using the key, here the encryption algorithm is absent, and the decryption key is essentially the neural network's coefficients, which must be determined through a training procedure. The proposed scheme can also be interpreted as associative memory, with the specified byte sequence located in one of its regions.

To enhance the degree of obfuscation, the optimal structure of the neural network would generate output data not as one large fragment of bytecode but sequentially, for example, byte by byte, after each subsequent byte is fed into the network. It is sufficient to accumulate bytes from the neural network's output in memory only until a complete command code is obtained, which, after being executed by the virtual machine, will be step-by-step replaced by the next code. Alternatively, if all virtual machine command codes are of the same length, the output should produce data of the corresponding bit width. This way, there will never be more than one virtual machine command stored in memory at any given time, further complicating the analysis of such code. Recurrent neural networks, particularly those with LSTM architecture, which are well-suited for processing sequences [28], best fit this description.

Block-wise processing can also be used, generating relatively short fragments of the output sequence (e.g., 16 bytes long) by either processing the same input sequence with several different neural networks or processing multiple parts of the input sequence with a single neural network trained to memorize several sequence pairs.

B. TRAINING PROCEDURE

The goal of the training process is to obtain a neural network that generates the predefined byte code only when the predefined hashed ID (obtained from user data and hardware identifiers) is presented on its input. For any other byte sequence, an arbitrary pseudo-random output should be generated.

Thus, the loss function for network training is expected to satisfy the following conditions:

- to be minimal for the case when input data is the correct hashed ID and output data precisely equals the predefined byte code;

- to be relatively small for the case when both input and output data are not predefined values (since a garbage-in – garbage-out behavior is correct in this case);

- to be big for other cases.

The proposed loss function is:

$$L(\boldsymbol{y}, \breve{\boldsymbol{y}}) = \begin{cases} \|(\boldsymbol{y} - \breve{\boldsymbol{y}})\|^2, & \breve{\boldsymbol{y}} = \boldsymbol{y}_p \\ \alpha \cdot \|(\boldsymbol{y} - \breve{\boldsymbol{y}})\|^2, & \breve{\boldsymbol{y}} \neq \boldsymbol{y}_p \end{cases}, \quad (1)$$

where $\boldsymbol{y_p}$ is the predefined output and α is a small constant (value of 0.05 was used in experiments). This setup leverages the ability of a neural network to act like associative memory: the neural network memorizes the mapping between the predefined hashed ID and the specific byte code, acting like an addressable memory that outputs the byte code when presented with the correct "address" (hashed ID), while for other inputs, the network does not attempt to memorize the mappings precisely, and due to the weak loss applied to random inputs, it naturally generates diverse, pseudo-random outputs. The network may still loosely memorize these input-output pairs, but with less precision than the predefined pair. Since the loss is scaled down for these pairs, some degree of memorization occurs, but it is less accurate and likely contains more errors. These secondary associations are flexible and error-prone, giving the appearance of pseudo-random behavior rather than exact mappings.

The first dense layer (see Fig. 2), used for hashing user data and hardware identifiers, has a sigmoid activation function. The last layer of the neural network being trained also uses a sigmoid activation function. Thus, the input and output of the network are modeled as sequences of floating-point numbers in the range [0, 1). To obtain the resulting bytecode, normalization is applied to outputs using multiplying by 256, followed by rounding to the nearest integer.

The training set is formed from pairs of input and output sequences, where input sequence can be the predefined hashed ID (with predefined output sequence as output) or a random sequence (with a random sequence as output). As memorizing the predefined input-output pair is the main goal, it should appear frequently in the training set to reinforce its importance. The training set should be big enough to ensure the network learns to generate varied, flexible outputs for all non-predefined cases, but not too big, so that the training time remains acceptable for practical implementation (possibly around a few dozen seconds). For our experiments, the training set consisted of 8 batches each containing 32 training samples, with 25% of predefined input-output pairs in each batch.

Random inputs are mainly generated with a uniform random generator, and partly (up to 10% of training set) are obtained with a permutation of the predefined hashed ID. All random outputs are generated with a uniform random generator.

Training is stopped once 100% accuracy is achieved for the pairs of predefined hashed ID and predefined byte code. This accuracy is calculated as the percentage of correctly reproduced symbols of the output sequence.

Typical figures for custom loss and custom accuracy of training a network consisting of LSTM layer with 128 units and output dense layer with 16 neurons are shown in Fig. 3.

Using this custom accuracy as a criterion for stopping neural network training leads to a satisfactory result in most cases, but sometimes training process may stop too early, when the predefined pair is perfectly memorized, but other attractors in output space have not been formed, so the network tends to output low variance data for any input except of predefined sequence. This is illustrated by Fig. 4, 5 and 6, where training results for a network with two dense and one LSTM layer are presented.

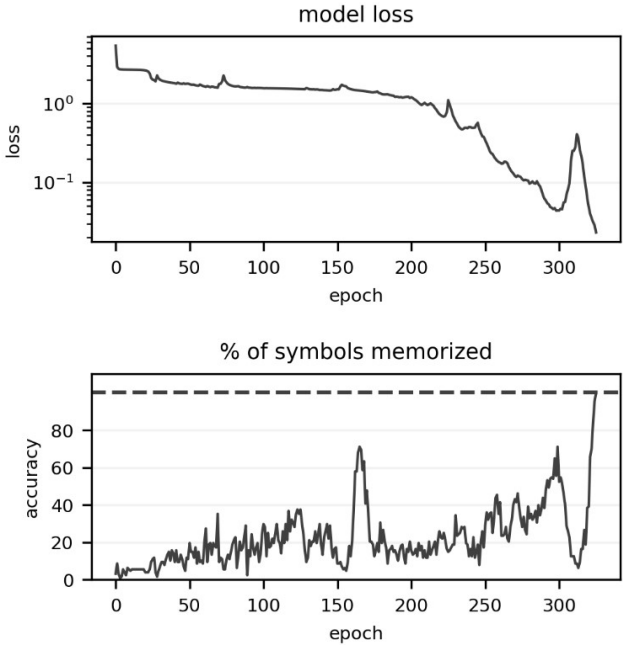

Figure 3. Custom loss and custom accuracy for a typical training process (N_{LSTM} = 128, α = 0.05)

Fig. 4 shows a very rapid increase in accuracy to 100% after 13 training epochs, while model loss and mean absolute error are still high enough.

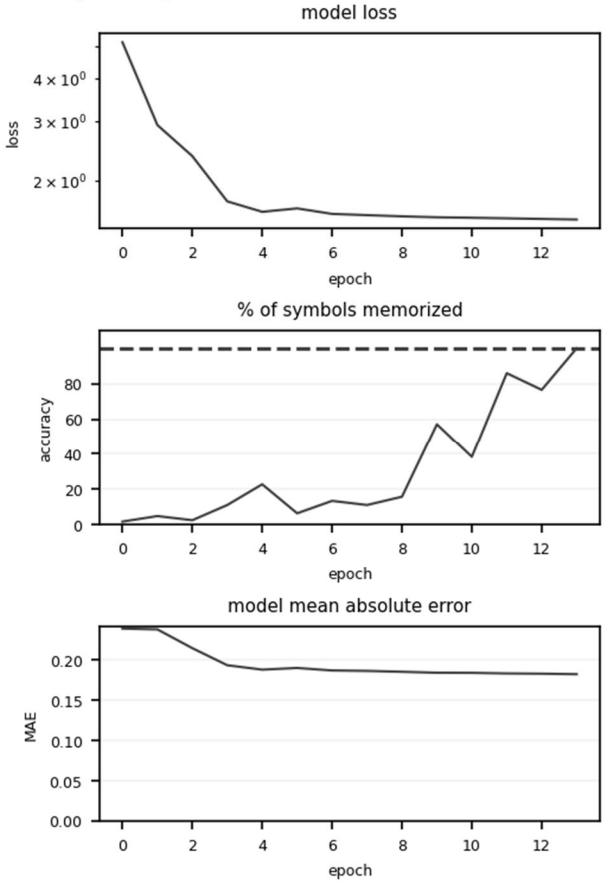

Figure 4. Example of too early training stop (N_{D1} = 256, N_{D2} = 128, N_{LSTM} = 64, α = 0.05)

Error distribution for output sequences after this training is shown in Fig. 5a. The error is calculated as the difference between the integer values in the output training samples and the floating-point values obtained by multiplying the network output by 256 before rounding to an integer. For the predefined sequence, error is in range (-0.5, +0.5) in both cases as expected, but for other sequences from training set, early stop led to wide error range of (-204.7, +194.54) with mean average value of 62.1 and almost uniform distribution (Fig. 5a). Analysis of model outputs shows that for both random input sequences and input sequences from the training set other than predefined sequence, the output data range is limited to 128±20, while when the input sequence resembles the predefined input sequence, even with 20% noise, the output data exhibits greater variance and more closely resembles the predefined output sequence. This is a highly undesirable phenomenon from the perspective of the cryptographic security of the proposed method. These features of the output data distribution are shown in Fig. 6a.

Fig. 5b shows a more successful training result, where, although the error range (- 113.1, +100.1) is rather large, the error distribution has a much more pronounced peak, and the mean absolute error is only 9.5.

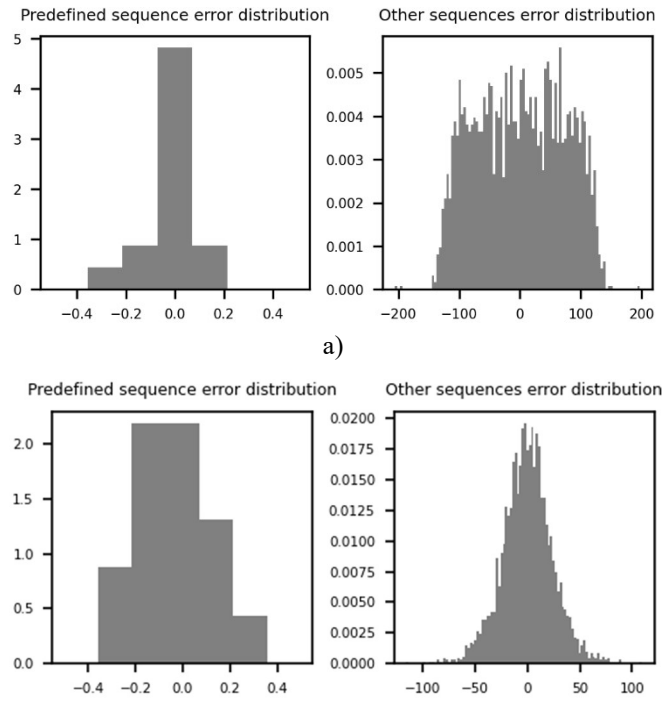

Figure 5. Error distribution: (a) – training stopped too early, (b) – training successful

Fig. 6 shows the mean values (gray diamonds), standard deviations (thick lines), and ranges (thin lines) for each of the 16 output symbols compared to the predefined output symbols (marked by x's). These results are obtained from processing 10,000 random input sequences with a uniform distribution by two neural networks with identical structure but different training result.

As shown in Fig. 6a, simply testing the neural network whose training ended too early by feeding it random input sequences allows an attacker to approximately determine the output symbols of the predefined sequence. The correlation coefficient between the predefined sequence and the sequence of maximum outlier values at each position is 0.98, while the average absolute difference between symbols of these two sequences is only 11. In contrast, if the mean average error for all sequences from the training set is sufficiently low after training the neural network, the output symbols exhibit a broader distribution (Fig. 6b). The range and standard deviation are sufficiently wide and nearly identical for all output symbols in this case, preventing an attacker from determining the approximate values of the predefined output sequence through simple statistical analysis.

A slight correlation is present even in this case between the predefined output sequence and the sequence of mean values. For the training result presented in Fig. 6b, the correlation coefficient is 0.11, but its absolute value varies from 0.01 to 0.34 for different training results with average of 0.18. However, this does not allow for determining the exact or even approximate values, but only a very rough shape of the sequence graph, which, without knowledge of the scale, does not provide any significant advantage. In any case, it may be advisable to repeat the training procedure if the absolute value of this correlation coefficient is greater than 0.2.

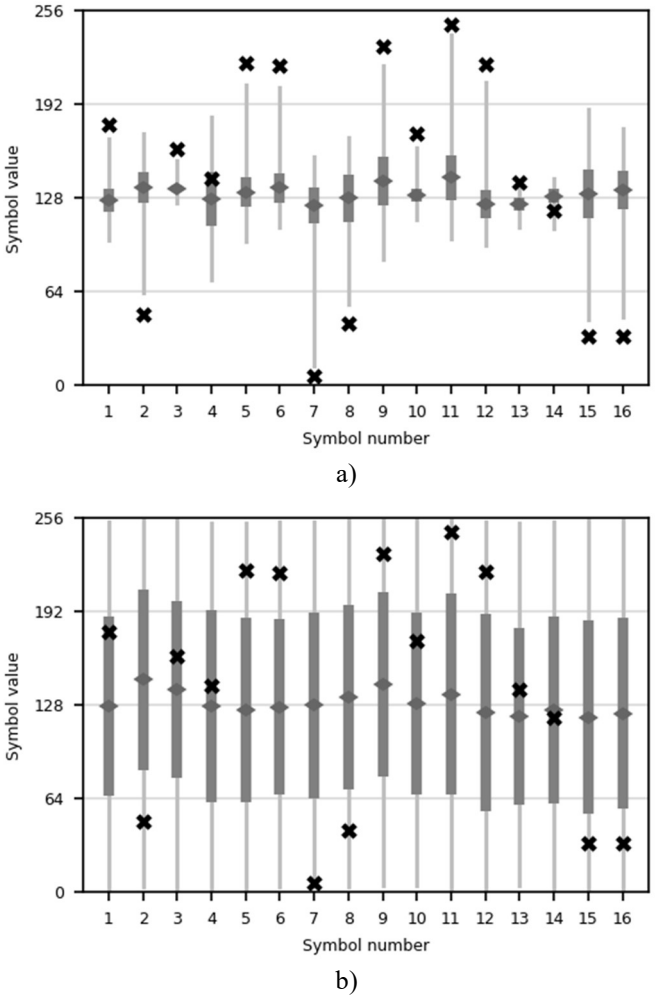

Figure 6. Output data distribution: (a) – training stopped too early, (b) – training successful

To prevent early stopping of training, it is proposed to use an additional criterion for terminating the training process – reaching the threshold value of the mean absolute error. Let the proportion of predefined pairs in the training set be β, then to obtain the mean absolute error no more than E_0 for predefined

output sequences and no more than E_1 for other sequences, the threshold value should be

$$E_T = \beta \cdot E_0 + (1 - \beta) \cdot E_1 \quad (2)$$

For instance, when 25% of predefined pairs are used in the training set, achieving a target mean absolute error of 10% of the data range (0, 255) requires a threshold value of $E_T = 0.075$. An example of a training process where applying this threshold prevented too early termination is shown in Fig. 7 (for a network with the same structure of two dense and one LSTM layer as above).

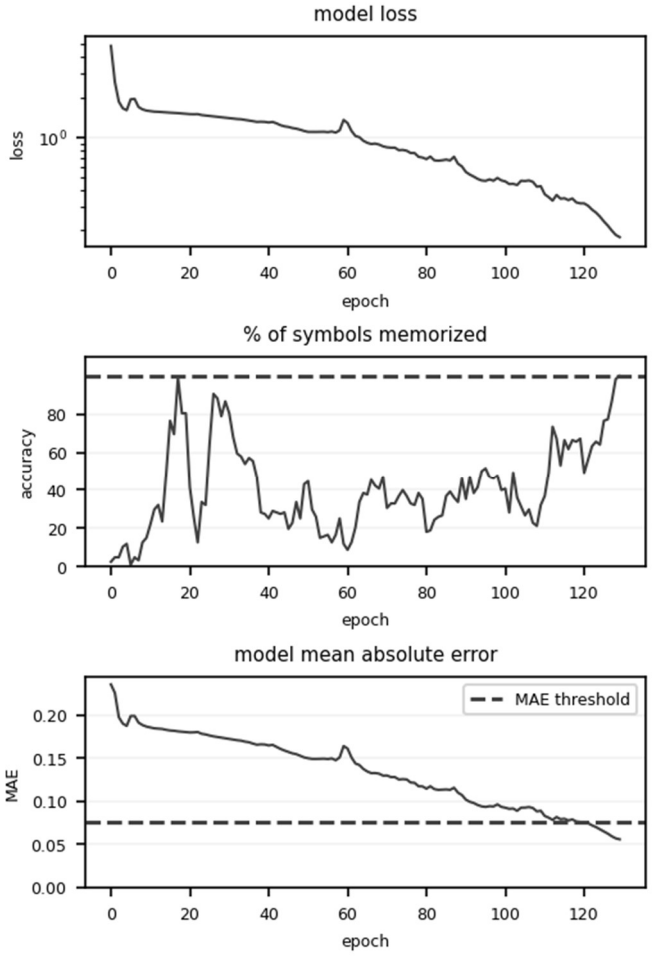

Figure 7. MAE threshold prevents too early training stop (N_{D1} = 256, N_{D2} = 128, N_{LSTM} = 64, α = 0.05)

In this case, 100% accuracy was achieved after the 18th epoch, however, the training continued until both the MAE and accuracy targets were met. Fig. 5b and 6b relate to this very training result.

Experiments showed that after reaching relatively high accuracy, a decrease in accuracy and an increase in loss occasionally occurred (see Fig. 3 near epochs 160 and 300). To mitigate this effect, the learning rate was adjusted as follows: at the end of each learning epoch, if the accuracy is higher than a given threshold (e.g. 75%) and current learning rate did not reach its lower threshold (e.g. 0.001), the learning rate is scaled by a factor of 0.75.

III. RESULTS

A. SELECTION OF THE OPTIMAL NEURAL NETWORK STRUCTURE

Several neural network configurations were evaluated for the task of memorizing symbol sequences. Key metrics considered in the analysis included the number of epochs and time needed to reach the target accuracy of 100%, the total number of trainable parameters and mean absolute error values for predefined sequence as well as for other sequences.

All configurations include dense output layer of 16 neurons with sigmoid activation function which is not specified in column "Configuration" (i.e. configuration "Dense (256)" means two dense layers, the first one with 256 and the second one with 16 neurons). All dense layers specified in the table use ReLU activation function, and all LSTM units use default activation configuration, namely hyperbolic tangent activation function for output and sigmoid activation function for the recurrent step. For model training, Adam optimizer was used with initial learning rate of 0.075 and learning rate adjustment with lower threshold of 0.001 as described in section II B. The mean absolute error threshold was set to 10% of the data range (E_T =0.075).

Each configuration was tested three times. The minimum and maximum numbers of epochs and average time to achieve the target accuracy are shown in Tables 1 and 2. MAE values are average of three tests and normalized to range (0, 255).

Table 1. Results of Neural Network Training Experiments (1 or 2 hidden layers)

Configuration	Total trainable para-meters	Number of epochs to achieve the target accuracy	Average training time, sec	Reached MAE	
				prede-fined	other
Dense (512)	16912	4927..7829	284.36	0.10	1.61
Dense (1024)	33808	2444..3549	126.93	0.10	1.42
Dense (2048)	67600	2070..2691	105.10	0.08	1.17
Dense (4096)	135184	1372..1798	72.68	0.08	1.32
Dense (8192)	270352	1469..2096	93.29	0.09	1.14
Dense (512) +Dense (256)	144144	749..1115	46.32	0.09	1.57
Dense (512) +Dense (512)	279568	247..534	23.71	0.09	1.64
Dense (1024) +Dense (512)	550416	256..497	30.98	0.06	1.60
Dense (1024) +Dense(1024)	1083408	277..381	51.55	0.10	1.98
Dense (2048) +Dense(1024)	2149392	424..638	137.34	0.09	2.34
LSTM (64)	17936	476..1369	85.41	0.10	12.69
LSTM (80)	27536	578..1216	81.13	0.10	9.14
LSTM (96)	39184	352..1062	70.21	0.09	6.13
LSTM (128)	68624	281..1213	83.36	0.09	6.59
LSTM (160)	106256	261..3044	135.17	0.11	6.23
Dense (64) + LSTM (64)	35152	334..696	28.96	0.11	3.53
Dense (128) + LSTM (64)	52624	259..386	17.44	0.09	3.82
Dense (256) + LSTM (64)	87568	185..395	16.75	0.11	5.67
Dense (128) + LSTM (128)	135824	220..326	14.69	0.08	2.02
Dense (256) + LSTM (128)	203536	117..207	12.91	0.07	3.46
Dense (192) + LSTM (192)	302032	200..299	17.79	0.08	1.63
Dense (256) + LSTM (256)	533776	169..249	21.54	0.10	1.44

Table 2. Results of Neural Network Training Experiments (3 or 4 hidden layers)

Configuration	Total trainable para-meters	Number of epochs to achieve the target accuracy	Average training time, sec	Reached MAE	
				prede-fined	other
Dense (64) + Dense(64) + LSTM (64)	39312	152..310	12.60	0.09	8.74
Dense (256) + Dense(128) + LSTM (64)	87696	120..224	10.85	0.10	18.55
Dense (128) + Dense(128) + LSTM (128)	152336	135..207	10.50	0.10	4.31
Dense (256) + Dense(256) + LSTM (128)	269328	117..133	10.89	0.08	5.22
Dense (128) + Dense(128) + LSTM (256)	417040	120..145	14.89	0.08	3.22
Dense (256) + Dense(256) + LSTM (256)	599568	107..173	16.05	0.09	2.61
Dense (64) + Dense(64) + LSTM (64) + LSTM (64)	72336	135..734	17.61	0.08	13.25
Dense (128) + Dense(128) + LSTM (128) + LSTM(128)	283920	90..631	14.01	0.10	9.15
Dense (256) + Dense(256) + LSTM (128) + LSTM(128)	400912	113..348	19.09	0.08	16.05

A total of 31 different configurations were analyzed. For neural networks with a homogeneous structure (one or two dense or one LSTM layer), the training time turned out to be significantly longer, especially for structures with only one dense layer. Increasing the number of neurons in such architectures allows reducing the training time only up to a certain limit; beyond this point, as the number of parameters increases, the training time starts to grow instead. This is illustrated in Fig. 8, which shows the dependence of training time on the total number of trainable network parameters. This pattern is also valid for mixed architectures, as shown in Fig. 9.

The notation "N1-N2-N3" in Figs. 8 and 9 represents the network structure, where N1 is the number of neurons in the first dense layer, N2 is the number of neurons in the second dense layer, and N3 is the number of units in the LSTM layer (e.g., 512-0-0 denotes a structure with one dense layer containing 512 neurons, 1024-1024-0 represents a structure with two dense layers, each containing 1024 neurons, and 0-0-160 represents a structure with a single LSTM layer with 160 units). As in tables above, output dense layer of 16 neurons is the same for all structures and is not mentioned in labels.

Most architectures with a single LSTM layer, even with a small number of units, provide significantly better performance than purely dense architectures with a comparable number of parameters. However, mixed architectures proved to be the most efficient overall (see Fig. 9).

Using two dense layers before the LSTM layer leads to a slightly shorter training time than for one dense layer, but further increasing the number of layers does not lead to an improvement in training time, as shown in the last three rows of Table 2, which present results for architectures with two dense and two LSTM layers.

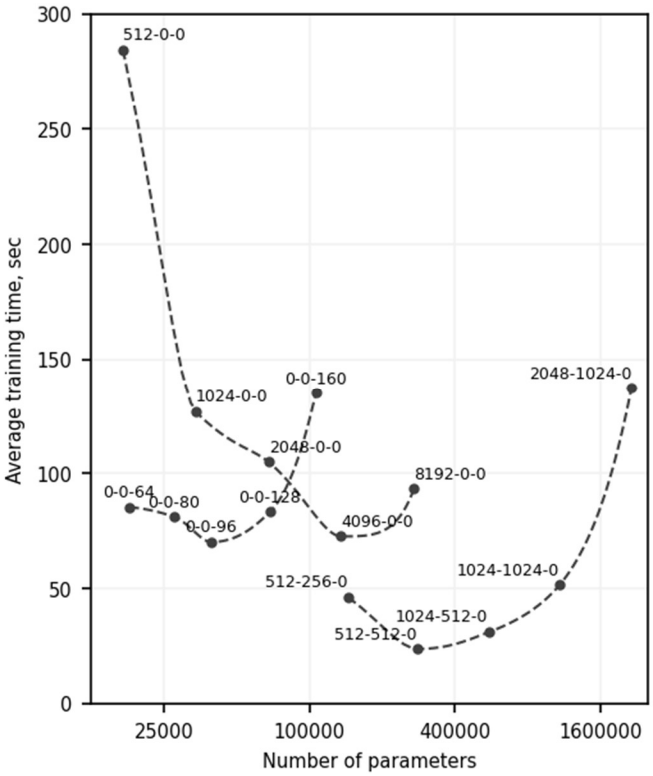

Figure 8. Average training time for homogeneous layer structures

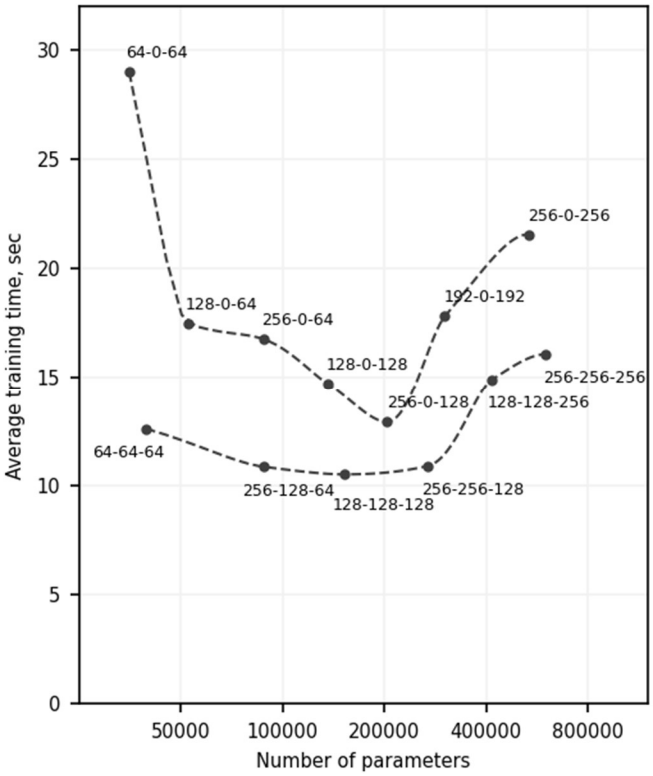

Figure 9. Average training time for mixed layer structures

The best-performing configurations were those with two dense layers and one LSTM layer, containing 128 or 256 neurons in the dense layers and 128 units in the LSTM layer, achieving 100% accuracy very quickly (average training time was about 11 sec). These configurations demonstrated high

stability and are considered optimal for the problem under consideration.

Experiments also showed that for the best configurations with 128 units in LSTM layer (labeled as "128-128-128" and "256-256-128" in Fig.9), the output layer size can be increased up to 36 neurons with only a slight increase in training time (Fig.10).

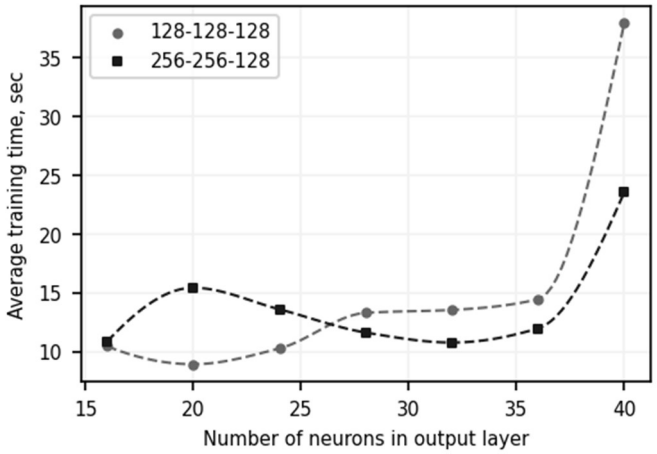

Figure 10. Average training time vs output layer size

This allows for memorizing longer bytecode sequences for each 16-byte input block and reduces the total size of activation data. For example, a bytecode sequence of length 108 can be produced using three neural network decoding blocks, each one with 16-byte input, two dense layers of 128 neurons, one LSTM layer of 128 units and output dense layer of 36 units, having 156916 trainable parameters per network, or 470748 parameters in total, or about 1.8 Mbytes of activation data.

B. ANALYSIS OF OUTPUT DISTRIBUTION PATTERNS

Analyzing the statistical behavior of output data generated from random input sequences with varying probability distributions provides insight into the method's security. If the attackers can decompile the code and reveal the network structure but does not have access to the valid activation code for the hardware data, their possible actions include neural network coefficients analysis and output distribution analysis. Ideally, network outputs should look like random noise for any input except of valid activation code, and neural network weights should not be correlated neither with the activation code nor the predefined output.

Fig. 11 shows the output distribution for a network with two dense and one LSTM layer (marked as "128-128-128" in previous section) with 36 output neurons that was obtained by feeding 10000 random input sequences to the network inputs and averaging output values (normalized to range [0,256) and rounded to integer) for all outputs. The input values had uniform (Fig. 11, a), truncated normal with standard deviation of 64 (Fig. 11, b), and beta distributions (Fig 11, c, d).

The output distribution is not uniform, but is unimodal, symmetrical in most cases and has sufficiently high entropy of 7.84…7.94, which is quite close to the entropy of the corresponding uniform distribution (8).

For asymmetrical beta input distribution, the output distribution is also slightly asymmetric, but skewness is very small: only 0.02 for a=2, b=5 and 0.04 for a=5, b=2.

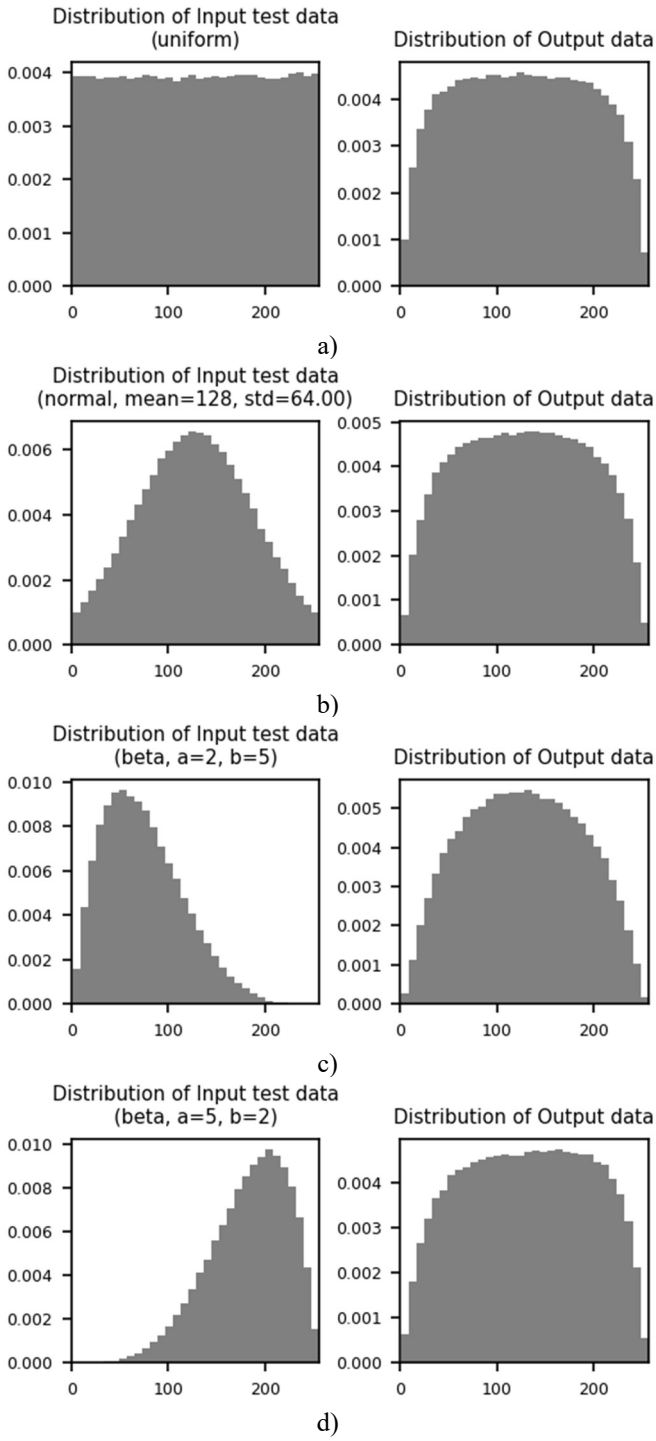

Figure 11. Output data distributions

Experiments with various activation functions in the output layer (tanh, linear, ReLU, and sigmoid with custom scaling) showed that the choice of activation function had little impact on the shape of the output distribution.

IV. COMPARISON WITH EXISTING PROTECTION METHODS

To assess the effectiveness of the proposed neural network–based protection method, it is compared with the main software protection techniques currently in use: classical white-box cryptography, virtualization-based obfuscation, and server-side license verification. Their key characteristics are summarized in Table 3.

Table 3. Comparison with existing approaches

Method	Resistance to Analysis		Hard-ware Binding	Transpa-rency of Key Represen-tation
	Static	Dyna-mic		
AES / DES White-Box Implemen-tations	High	Medi-um	Optional	Explicit (key tables or encoded constants)
Virtualization / Obfuscation	High	High	No	Explicit in bytecode form
Server-Side License Verification	Medi-um	High	Yes	Hidden on server
Proposed Neural-Network-Based Method	Very High	High	Yes	Implicitly distributed in network weights

Classical white-box cryptographic systems rely on statically encoded key tables or algebraic transformations. Although efficient in execution, such structures may be vulnerable once the lookup data are reconstructed from memory, revealing partial or full key information.

Virtualization-based obfuscation provides stronger defense against static analysis by converting protected code into virtual instructions executed by a custom interpreter. This makes reverse engineering considerably more difficult but still allows potential runtime tracing or emulation attacks that can gradually reconstruct program logic.

Server-assisted license verification adds remote control of activation and ensures that core validation routines are hidden from the client side. However, its security depends on continuous network availability and the integrity of communication channels; spoofing or imitation of server responses can undermine its reliability.

In contrast, the proposed neural network-based method eliminates explicit key representation altogether. The mapping between input identifiers and the generated bytecode is encoded implicitly within the network's parameters, forming a distributed structure that cannot be directly extracted or reconstructed. This provides a very high level of static resistance and strong defense against dynamic and differential attacks. The approach also ensures individualization, since each model is trained using a unique combination of user and hardware identifiers.

Modern protection frameworks such as Themida [29], Safengine [30], and Enigma Protector [31] rely primarily on deterministic code transformation, static bytecode virtualization, or anti-debugging mechanisms. While effective, these methods depend on predefined transformation rules that can, in principle, be analyzed and emulated. The proposed neural network-based system extends this paradigm by introducing non-deterministic, learned transformations that generate unique activation models for every user, thus making automated cracking, model replication, or parameter inference practically infeasible.

What principally distinguishes the proposed approach from traditional virtualization-based protection is the encryption (obfuscation) phase itself. In our system, this stage corresponds to training the neural network, whereas in classical white-box and virtualization frameworks it is the deterministic code-generation step. Therefore, the following analysis focuses specifically on this phase – the computational effort and memory footprint required to produce or transform the protected code.

In early table-based white-box implementations such as [2], the authors already reported that their AES variant was "significantly larger and slower" than the standard algorithm. Later evaluations, including [32], measured this difference by evaluating several white-box AES variants in terms of execution time per 128-bit block and memory usage. Their results indicate that conventional AES is several orders of magnitude faster, while white-box realizations occupy hundreds of kilobytes to gigabytes and require tens of microseconds per block. For example, the optimized method in [32] reaches about 27.9 µs per block with roughly 0.6 MB of code, whereas larger parameter sets easily exceed hundreds of MB. Because AES is symmetric, encryption and decryption show nearly identical timings in those systems.

On the other hand, the proposed neural-network approach exhibits a fundamentally different performance profile. The encryption phase (training) occurs once during model registration and is not repeated during runtime. The decryption phase (inference) is extremely lightweight, involving only a forward pass through the network without large lookup tables; its complexity scales with the number of parameters rather than with memory accesses. Typical model sizes remain within a few hundred kilobytes – comparable to or smaller than recent compact white-box schemes such as WAS [33], which reports about 128 KB memory usage at similar space-hardness levels.

Overall, the proposed method provides protection within a memory footprint characteristic of compact white-box AES implementations while avoiding their extensive table lookups and deterministic structure. This results in predictable and memory-efficient runtime performance, comparable to compact white-box AES implementations, while maintaining higher structural diversity and resistance to code extraction.

V. CONCLUSIONS

Protecting software against unauthorized use and malicious activities requires a comprehensive approach involving both hardware and software mechanisms. The proposed scheme ensures robustness against attacks by leveraging obfuscation, encryption, and the intrinsic variability of neural network-based bytecode generation. Each step in the activation and verification process introduces layers of complexity that significantly hinder unauthorized access or replication of the protected code.

The method binds the activation process to the user-specific information and unique hardware identifiers of the target machine, ensuring that the activation code and resulting bytecode are inherently tied to the specific environment. This prevents the use of a valid activation code on a different machine. The random initialization of the network's parameters ensures that the activation code is unpredictable and difficult to replicate or reverse-engineer. Even if an attacker gains access to one valid activation code, the neural network's pseudorandom behavior and variability in output for other inputs prevent the attacker from generalizing the result to other environments or generating usable codes for unauthorized installations. The virtual machine's role in interpreting the bytecode adds an additional layer of abstraction and security, as the bytecode must be executed in a specific and controlled manner to achieve successful program operation. Finally, the

continuous re-verification of the activation code during the program's lifecycle ensures that even if an attacker temporarily bypasses protection, their access is limited to the immediate session. Each subsequent verification follows the same rigorous process, making persistent unauthorized access impractical.

This layered security approach, combining hardware-specific binding, neural network variability, and unique bytecode generation, ensures that the proposed method remains resilient against common attack vectors and significantly raises the barriers for potential adversaries.

References

[1] C. Collberg, C. Thomborson, D. Low, *A Taxonomy of Obfuscating Transformations*, Technical Report, University of Auckland, 1997, 36 p.

[2] S. Chow, P. Eisen, H. Johnson, P.C.V. Oorschot, "White-box cryptography and an AES implementation," *Selected Areas in Cryptography*, vol. 2595, Springer, Berlin, Heidelberg, 2003, pp. 250–270. https://doi.org/10.1007/3-540-36492-7_17

[3] R. Rolles, "Unpacking virtualization obfuscators," *WOOT'09: Proceedings of the 3rd USENIX Conference on Offensive Technologies*, Montreal, Canada, August 10–14, 2009. Available at: https://www.usenix.org/legacy/event/woot09/tech/full_papers/rolles.pdf

[4] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, issue 1, article 4, pp.1-37, 2016. https://doi.org/10.1145/2886012

[5] H. Fang, Y. Wu, S. Wang, Y. Huang, "Multi-stage binary code obfuscation using improved virtual machine," *Proceedings of the International Conference on Information Security*, Xi'an, China, October 26-29, 2011, pp. 168–181. https://doi.org/10.1007/978-3-642-24861-0_12

[6] I. Meraouche, S. Dutta, H. Tan, K. Sakurai, "Neural networks-based cryptography: A survey," *IEEE Access*, vol. 9, pp. 124727–124740, 2021. https://doi.org/10.1109/ACCESS.2021.3109635

[7] A. El-Zoghabi, A.H. Yassin, H.H. Hussien, "Survey report on cryptography based on neural network," *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, issue 12, pp. 456–462, 2013.

[8] C.K. Chan, C.K. Chan, L.P. Lee, L.M. Cheng, "Encryption system based on neural network," in: R. Steinmetz, J. Dittman, M. Steinebach (Eds.), *Communications and Multimedia Security Issues of the New Century*, Springer, 2001, pp. 117–122. https://doi.org/10.1007/978-0-387-35413-2_10

[9] D. Guo, L.M. Cheng, L.L. Cheng, "A new symmetric probabilistic encryption scheme based on chaotic attractors of neural networks," *Applied Intelligence*, vol. 10, pp. 71–84, 1999. https://doi.org/10.1023/A:1008337631906

[10] N. Liu, D. Guo, "Security analysis of public-key encryption scheme based on neural networks and its implementation," *Proceedings of the 2006 IEEE International Conference on Computational Intelligence and Security*, Guangzhou, China, November 3-6, 2006, pp. 1327–1330. https://doi.org/10.1109/ICCIAS.2006.295274

[11] Y. Fu, J. Fu, J. Wei, Encryption and decryption using deep neural network, in: J.-L. Kim (Ed.) *Machine Learning and Artificial Intelligence*, IOS Press, 2023, pp. 9–15. Available at: https://ebooks.iospress.nl/doi/10.3233/FAIA230762

[12] K. Kumar, S. Tanwar, and S. Kumar, "MANC: A masked autoencoder neural cryptography based encryption scheme for CT scan images," *MethodsX*, vol. 12, p. 102738, 2024. https://doi.org/10.1016/j.mex.2024.102738

[13] A. Radhakrishnan, M. Belkin, and C. Uhler, "Overparameterized neural networks implement associative memory," *Proceedings of the National Academy of Sciences*, vol. 117, no. 44, pp. 27162-27170, 2020. https://doi.org/10.1073/pnas.2005013117

[14] H. He, Y. Shang, X. Yang, Y. Di, J. Lin, Y. Zhu, W. Zheng, J. Zhao, M. Ji, L. Dong, and N. Deng, "Constructing an associative memory system using spiking neural network," *Frontiers in Neuroscience*, vol. 13, article 650, pp.1-15, 2019. https://doi.org/10.3389/fnins.2019.00650

[15] Y. Tay, V. Tran, M. Dehghani, J. Ni, D. Bahri, H. Mehta, Z. Qin, K. Hui, Z. Zhao, J. Gupta, and T. Schuster, "Transformer memory as a differentiable search index," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21831-21843, 2022.

[16] I. Kanter, W. Kinzel, "Neural cryptography," *Proceedings of the 9th International Conference on Neural Information Processing*, Singapore, November 18-22, 2002, vol.3, pp.1351-1354

[17] T. Godhavari, N. Alamelu, R. Soundararajan, "Cryptography using neural network," *Proceedings of the Annual IEEE India Conference - Indicon*, Chennai, India, 11–13 December 2005.

[18] S. Jeong, C. Park, D. Hong, C. Seo, and N. Jho, "Neural cryptography based on generalized tree parity machine for real-life systems," *Security and Communication Networks*, article ID 6680782, pp. 1-12, 2021. https://doi.org/10.1155/2021/6680782

[19] M. Abadi, D.G. Andersen, "Learning to protect communications with adversarial neural cryptography," *arXiv*:1610.06918 [cs.CR], pp. 1–15, 2016. https://doi.org/10.48550/arXiv.1610.06918

[20] E.A. Hagras, S. Aldosary, H. Khaled, and T.M. Hassan, "Authenticated public key elliptic curve based on deep convolutional neural network for cybersecurity image encryption application," *Sensors*, vol. 23, issue 14, p. 6589, 2023. https://doi.org/10.3390/s23146589

[21] M.C. Wøien, F.O. Catak, M. Kuzlu, and U., Cali, "Neural networks meet elliptic curve cryptography: A novel approach to secure communication," *arXiv*:2407.08831 [cs.CR], pp.1-8, 2024. https://doi.org/10.48550/arXiv.2407.08831

[22] E. Volna, M. Kotyrba, V. Kocian, M. Janosek, "Cryptography based on neural network," *Proceedings of the 26th European Conference on Modeling and Simulation*, Koblenz, Germany, May 29 – June 1, 2012, pp. 386–391. https://doi.org/10.7148/2012-0386-0391

[23] R.R. Al-Nima, L. Muhanad, S.Q. Hassan, "Data encryption using backpropagation neural network," *IRAQI Academic Scientific Journals*, vol. 15, no. 2, pp. 112–117, 2009.

[24] R.A. Zitar, H. Hussain, "Mirroring neural network approach for encryption/decryption of data," *ICIC Express Letters*, vol. 13, no. 12, pp. 1057–1064, 2019.

[25] K. Shihab, "A backpropagation neural network for computer network security," *Journal of Computer Science*, vol. 2, no. 9, pp. 710–715, 2006. https://doi.org/10.3844/jcssp.2006.710.715

[26] R.K. Munkulu, V. Gnanam, "Neural network-based decryption for random encryption algorithms," *Proceedings of the 2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*, Hong Kong, China, 20-22 August, IEEE, 2009, pp. 603–605. https://doi.org/10.1109/ICASID.2009.5277002

[27] I.C. Lin, H.H. Ou, and M.S. Hwang, "A user authentication system using back-propagation network," *Neural Computing & Applications*, vol. 14, pp. 243-249, 2005. https://doi.org/10.1007/s00521-004-0460-x

[28] G. Van Houdt, C. Mosquera, G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5929–5955, 2020. https://doi.org/10.1007/s10462-020-09838-1

[29] Themida - Advanced Windows software protection system. [Online]. Available at: https://www.oreans.com/Themida.php

[30] SafeEngine overview. [Online]. Available at: https://www.safengine.com/en-us/features/overview

[31] Enigma Protector - Software Licensing and Protection System. [Online]. Available at: https://enigmaprotector.com/en/about.html

[32] A. Battistello, L. Castelnovi, T. Chabrier, "Enhanced encodings for white-box designs," *Proceedings of the International Conference on Smart Card Research and Advanced Applications*. Cham: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-97348-3_14

[33] Y. Yang, Y. Zhai, H. Dong, Y. Zhang, "WAS: improved white-box cryptographic algorithm over AS iteration," *Cybersecurity*, vol. 6, issue 1, p. 56, 2023. https://doi.org/10.1186/s42400-023-00192-7

***Victor ROVINSKYI** received a specialist diploma in radio engineering from the Lviv Polytechnic University in 1996. He obtained a PhD degree from the Ivano-Frankivsk National Technical University of Oil and Gas in 2004. He is an associate professor at the Department of Computer Science and Information Systems at the Vasyl Stefanyk Precarpathian National University. His research interests include information protection technologies, automation and robotics, and digital signal processing for professional audio.*

Olga Yevchuk *received an engineering degree in 1999 and PhD degree in 2005 from Ivano-Frankivsk National Technical University of Oil and Gas. She worked in this university till 2023 and now is a software engineer at Softjourn Ukraine. Her research interests include machine learning, digital signal processing and oil equipment diagnostics.*

Yurl STRILECKYI *received an engineering degree in 1995 from Ivano-Frankivsk National Technical University of Oil and Gas. He defended his PhD dissertation in 1999 and his doctoral dissertation in 2018. Since 2022, he has been a professor at the Department of Information and Telecommunication Technologies and Systems at the same university. His research interests include digital signal processing, microprocessor systems, embedded technologies, robotics, and methods for assessing the technical condition of structures. He is the author of over 100 scientific publications and a co-author of eight patents. He actively participates in the development of high-tech industrial automation systems and the Internet of Things. He teaches courses in electronics, microprocessor technology, and data processing.*