Cryptanalytic Extraction of Recurrent Neural Network Models

Longxiang Wei[1,2,3], Hao Lei[1,2,3], Xiaokang Qi[1,2,3], Xiaohan Sun[1,2,3], Lei Gao[1], Kai Hu[1,3,4,5], Wei Wang[1,2,3,4], and Meiqin Wang[1,2,3] (✉)

[1] School of Cyber Science and Technology, Shandong University, Qingdao, Shandong, China
longxiangwei@mail.sdu.edu.cn,leihao@mail.sdu.edu.cn,xiaokangqi@mail.sdu.edu.cn, xhansun@mail.sdu.edu.cn,leigao@sdu.edu.cn,kai.hu@sdu.edu.cn, weiwangsdu@sdu.edu.cn,mqwang@sdu.edu.cn
[2] State Key Laboratory of Cryptography and Digital Economy Security, Shandong University, Qingdao, 266237, China
[3] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan, China.
[4] Quan Cheng Shandong Laboratory, Jinan, China
[5] Suzhou Research Institute, Shandong University, Suzhou, 215123, China

Abstract. In recent years, neural network extraction has been studied with cryptographic techniques, since Carlini et al.'s pioneering work proposed at CRYPTO 2020. Most research has focused on simple fully connected network (FCN) models, with limited attention given to more complicated recurrent neural network (RNN) models. However, RNN models are dominant in fields such as natural language processing and speech recognition. Exploring the vulnerability of RNN models to extraction attacks is not only methodologically significant but also reveals an attack surface broader in scope and higher in real-world impact.

In this work, for the first time we propose a series of cryptanalytic extraction attacks against RNN models under both the raw-output (S5) and hard-label (S1) scenarios. Our attack selects inputs to establish an equivalence between the RNN and shallow FCN models. Since the parameters of these equivalent models are entangled with neuron permutations and scaling factors, they must be aligned before reuse. In the S5 scenario, we construct an equivalent FCN model and apply permutation and scaling alignment methods to enable parameter reuse. In the S1 scenario, we establish an equivalence between one RNN and two FCN models, and propose permutation search, accuracy enhancement and sign search methods to address the challenges of hard-label scenarios.

In the S5 scenario, we recover the parameters of five RNN models with different configurations, while in the S1 scenario, we recover those of two RNN models, and in both cases the models reach depths of up to 1024 layers. To the best of our knowledge, this is the first time that model extraction attacks have been extended from networks with fewer than 10 layers to networks with thousands of layers.All experiments are completed on a PC within two hours.

Keywords: ReLU-based Neural Networks · Functionally Equivalent Extraction · RNN · Hard-Label.

1 Introduction

Deep neural networks are a class of machine learning models characterized by multiple hidden layers, enabling them to capture complex, high-dimensional patterns and achieve state-of-the-art performance across various domains, including computer vision, natural language processing, and others. Common architectures include fully connected network (FCN) models for general-purpose function approximation [23], recurrent neural network (RNN) models for sequential data modeling [24], and convolutional neural network (CNN) models for spatial data processing [16], and so on.

Training such models necessitates considerable computational resources and extensive datasets; as a result, the resulting models constitute valuable intellectual property. Consequently, deep neural network models emerged as prime targets for model extraction attacks and adversarial techniques were designed to reconstruct a functionally equivalent network via black-box access. Based on the type of information accessible to the adversary, the threat model can be broadly classified into five distinct scenarios [3]. In the S1 scenario, the adversary can only access the most likely class label associated with a given input (the hard-label setting). In the S2 scenario, the adversary is provided with the single most likely class label together with its probability score; in the S3 scenario, with the top-k class labels along with their probability scores; and in the S4 scenario, with the complete set of class labels and their associated probability scores. In the S5 scenario, the adversary is granted access to the raw output of the neural network prior to the normalization phase, which provides substantially more information. Previous work has primarily focused on the most challenging scenario, S1, as well as the simplest scenario, S5.

Several studies [1,4,12,17] have examined FCN models in the S5 scenario. A significant breakthrough was reported in [4], which introduced a layer-by-layer model extraction approach consisting of two sequential stages: first, recovering the absolute-form values of the layer's parameters in a deep neural network, and then determining their corresponding signs. Experimental evaluations on multiple neural network architectures demonstrate that this attack can recover models with a precision improvement of 2^{20} while requiring $100\times$ fewer queries compared to prior state-of-the-art methods. However, in the general case, the sign recovery method in [4] requires exponential computational effort, Canales et al. [1] introduced three novel sign-recovery techniques: system of equations, the neuron wiggle method, and the last hidden layer technique, in order to achieve sign recovery only a polynomial number of queries and polynomial time. At NeurIPS 2024, Foerster et al. [12] introduced a unified framework that improves the efficiency of weight sign extraction by up to $14.8\times$ through the identification of signs that are easier or harder to extract. They further show that neurons with signs that are harder to extract cannot be recovered using the neuron

wiggle method. Moreover, they demonstrate that extracting the absolute values of parameters (i.e., the signature), rather than merely their signs, constitutes the primary bottleneck in high-fidelity model recovery. Liu et al. [17] analyzed the challenges of signature recovery in deeper networks, particularly the impact on merging and rank deficiency, and proposed concrete algorithmic solutions. They improved the signature recovery method and performed an 8-layer non end-to-end attack to validate their enhanced signature recovery approach.

As mentioned in [2,3,12,17], the end-to-end model extraction relies on previously recovered results to extract subsequent layers. The model extraction, which is not end-to-end, requires utilizing the preceding parameters from the original model during the recovery of each layer's parameters. Aside from the three-layer end-to-end model extraction results provided by [4], the attacks of [1,17] on the 8-layer model are not end-to-end. Canales et al. [1] presented an 8-layer model extraction attack that is not end-to-end, as they only recovered the sign of the parameters rather than the signatures of the parameters. Liu et al. [17] also demonstrated 8-layer model extraction results, which are not end-to-end. The experiments in [17] failed due to accumulated floating-point imprecision errors in the end-to-end setting. As a result, they assumed that when recovering one layer, the preceding layers were perfectly recovered. Specifically, they performed model extraction attacks to recover the parameters of each layer, directly using the real parameters of the preceding layers to avoid the error issues.

While prior work has primarily examined FCN models in the S5 scenario, several studies [2,3,5,14] have also investigated them in the S1 setting. At ASIACRYPT 2024, Chen et al. [5] presented the first theoretical attack on ReLU-based networks in the S1 scenario, validated on MNIST and CIFAR-10, and achieved extraction for networks with over 10^5 parameters within hours on a single CPU core. However, the extraction method requires a polynomial number of queries but an exponential execution time. To overcome these limitations, Carlini et al. [3] introduced a dual point analysis method that recovers both the signatures and signs by exploiting the geometry of decision boundaries, enabling polynomial-time cryptanalytic extraction of FCN models. The process of recovering signatures primarily involves identifying dual points, clustering these points, and recovering the parameters based on the resulting clusters. Since the attack in [3] recovered the parameters of all layers except the output layer, due to the absence of ReLU activations, Canales et al. [2] presented a technique that enables the recovery of the output layer in the S1 scenario. In 2025, Ito et al. [14] propose a cross-layer extraction method for identifying persistent neurons that are consistently active, which reduces query complexity and mitigates the limitations of existing model extraction approaches.

We observe that existing studies have predominantly focused on FCN models, while investigations into other neural network architectures remain scarce. At EUROCRYPT 2024, Carlini et al. [1] argued that their attack method can be directly applied to CNN models, since CNN models can be regarded as a special case of FCN models in the S5 scenario; however, they did not provide an actual attack demonstration. In 2024, Coqueret et al. [9] explore how physical side-

channel traces can be leveraged to identify critical points and infer the signs of individual neurons, thereby successfully achieving model extraction attacks on CNN models with a larger number of parameters. Prior attacks on RNN model [25] adopted a distillation-style method, extracting a smaller-scale RNN model from a long short-term memory network model; however, the extracted model is not functionally equivalent [4] to the original. This raises the question of whether cryptanalytic extraction techniques can be generalized to architectures beyond FCN. Inspired by [1,2,3,4], we investigate how to recover functionally equivalent RNN models by leveraging their inherent architectural properties.

Our contributions. We present the first model extraction attack targeting RNN models in both the S1 and S5 scenarios. To achieve this, we introduce a model equivalence transformation method that reformulates the process of recovering RNN parameters into FCN models. In the S5 scenario, we construct an equivalent three-layer FCN model. In the S1 scenario, where parameter recovery is more challenging, we construct new equivalents that transform the RNN model into two equivalent FCN models with a maximum depth of two. After constructing the equivalent FCN models, we employ scenario-specific methods to make the recovered parameters reusable. Due to the aforementioned techniques, we are able to recover much deeper RNN models in practice.

- **Attack techniques in the logit-output setting (S5).** To make the parameters reusable, we adjust the recovered parameters to ensure that the permutation and scaling of neurons match those of the previous layer. The permutation alignment method focuses on aligning two adjacent recovered layer's weight by transforming the second matrix to match the first sign informuation, in order to compute the necessary column transformations matices. This method leverages both individual and aggregate sign information from rows and columns to compute candidate filters, while the overall algorithm runs in polynomial time. The scaling alignment method computes the ratio between the scaling of the preceding layer and the current layer to be adjusted, then applies this ratio to the rows of the parameter matrix and the bias of the current layer to ensure consistency in neuron scaling across layers.
- **Attack techniques in the hard-label setting (S1).** The permutation search strategy is designed to adjust the row permutation. By leveraging the sign information of the output-layer and first-layer biases, this search strategy looks for the correct row permutation of the intermediate-layer parameters that is valid based on dual points. When the permutation is fixed, the required scaling factors can be computed from the output layers of the two equivalent FCN models. To improve the accuracy of parameter recovery, we propose a method that filters dual points within each cluster by selecting those whose corresponding neuron values are farther from zero. For both RNN models, this method reduces the recovery error by approximately one order of magnitude and achieves accuracy close to that of the S5 scenario. According to [12], neurons with signs that are harder to extract cannot be recovered using the neuron wiggle method. Therefore, we propose a more robust sign search method to verify parameter recovery for RNN models.

– **Results.** In the S5 scenario, our end-to-end attack successfully extracts RNN models with depths of up to 1024 layers, as shown in Table 1. This result significantly surpasses the 3-layer recovery on FCN models reported in [4], as well as the 8-layer non-end-to-end recovery reported in [17]. In the S1 scenario, we further demonstrate end-to-end extraction of a 1024-layer RNN model, as shown in Table 1, exceeding the 2-layer recovery on FCN models reported in [5] and the 4-layer non-end-to-end recovery on FCN models reported in [3].

Table 1: Attack Results of RNN Models under the S5 and S1 Scenarios. The evaluation metrics (ε, δ) are defined in Definitions 4. Depth refers to the number of activation functions traversed from the first input to the output.

Structures	Type	Depth	Scenario	$(\epsilon, 0)$	$(\epsilon, 0.05)$	Reference
$50-25^{40}-1$	RNN	40	S5	$2^{-16.7}$	$2^{-20.9}$	This work
$50-16^{100}-1$	RNN	100	S5	$2^{-16.6}$	$2^{-20.9}$	This work
$50-16^{1024}-1$	RNN	1024	S5	$2^{-16.2}$	$2^{-20.9}$	This work
$100-16^{40}-1$	RNN	40	S5	$2^{-16.5}$	$2^{-20.5}$	This work
$800-16^{40}-1$	RNN	40	S5	$2^{-16.1}$	$2^{-19.8}$	This work
$50-16^{40}-1$	RNN	40	S1	$2^{-16.9}$	$2^{-19.9}$	This work
$50-16^{1024}-1$	RNN	1024	S1	$2^{-14.7}$	$2^{-18.8}$	This work

Outline. In Section 2, we briefly introduce an overview of the most relevant works. In Section 3, we describe complete attack pipelines and the relevant techniques for RNN models in the S5 scenario. In Section 4, we present complete attacks against RNN models and improvements to parameter recovery techniques in the S1 scenario. In Section 5, we describe experiments on training RNN models, and evaluate the recovered models in both the S1 and S5 scenarios. In Section 6, we conclude the paper.

2 Preliminaries

2.1 Notations and Definitions

In this section, we briefly review the notations and definitions used for model extraction attacks on FCN models in [1,3,4].

Definition 1. *Let $f_\theta(x)$ denote a k-layer neural network, parameterized by θ, that maps elements from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$. The mapping f is defined as a sequential composition of linear transformations f_j interleaved with a nonlinear, component-wise activation function σ:*

$$f = f_{k+1} \circ \sigma \circ \cdots \circ \sigma \circ f_2 \circ \sigma \circ f_1.$$

For a k-layer neural network model, the depth is defined as the number of hidden layers, excluding the input and output layers. The notation d_0 denotes the number of neurons in the input layer, d_{k+1} denotes the number of neurons in the output layer, and d_i with $1 \leq i \leq k$ denotes the number of neurons in the i-th hidden layer. The neural network models consider neural networks where $\mathcal{X} = \mathbb{R}^{d_0}$ and $\mathcal{Y} = \mathbb{R}^{d_{k+1}}$. We assume that floating-point representations exactly encode all elements of $\mathbb{R}$.

Definition 2. *The j-th layer f_j of a neural network is defined by the affine transformation*

$$f_j(x) = W^{(j)}x + b^{(j)},$$

where the parameter matrix $W^{(j)} \in \mathbb{R}^{d_j \times d_{j-1}}$ has dimensions $d_j \times d_{j-1}$, and the bias vector $b^{(j)} \in \mathbb{R}^{d_j}$ is d_j-dimensional.

While representing each layer f_j as a full matrix multiplication, commonly referred to as fully connected, is the most general formulation, such an architecture is termed as an FCN model, as illustrated in Fig. 1.

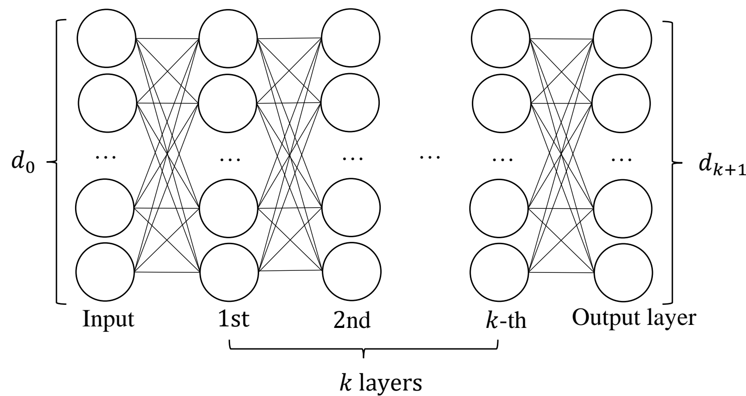


Fig. 1: The architecture of a k-layer FCN model.

Definition 3. *The neurons $\{n_i\}_{i=1}^{N}$ are functions that receive an input and apply the activation function σ. There are a total of $N = \sum_{j=1}^{k} d_j$ neurons.*

In this paper, we focus exclusively on the ReLU activation function [22], defined as $\sigma(x) = \max(x, 0)$.

To generalize the notation, an architecture $[d_0, \ldots, d_{k+1}]$ can equivalently be expressed as $d_0 - \cdots - d_{k+1}$. When consecutive layers share the same dimensionality, an exponential notation may be employed for conciseness. For example, $20 - 10^3 - 1$ denotes the architecture $20 - 10 - 10 - 10 - 1$.

2.2 Functionally Equivalent Extraction

In this section, we revisit differences between the recovered parameter matrices and the original ones using the method propsed by [1,2,3,4] in the S1 and S5 scenario. For details on model extraction, please refer to [1,2,3,4].

In the S5 scenario, the signature recovery method proposed by [4], which utilizes the properties of the activation function $\sigma(x)$, where the second derivative at $x = 0$ is non-zero and zero elsewhere, includes the following three steps. First, critical points are identified, where the value of a specific neuron is zero. Second, by making queries near the critical point while keeping the signs of other neurons unchanged, we can construct a system of equations to solve for the ratio relationship between the parameters of this neuron. The recovered parameters in this neuron differ from the original parameters due to an unknown scaling factor. Third, merge the parameters if they are proportional. Since the recovered parameters' correspondence to neurons in the original model is unknown, the difference between the recovered neurons and the original neurons is induced—the permutation of neurons.

In the S1 scenario, the dual points proposed by [3] replace the critical points, as we cannot obtain the model's raw output. The dual points are not only critical points but also specific inputs that make the scores of the two highest-scoring classes arbitrarily close. The signature recovery method proposed by [3] in the S1 scenario also relies entirely on critical points, which causes the recovered parameters in all layers to differ from the original ones due to the permutation and scaling of neurons in both the S1 and S5 scenarios.

The recovered parameter matrix, which exhibits a ratio related to the bias with the recovered bias set to zero, also results in a discrepancy between the recovered output layer in the S1 scenario and the original one.

In the following, $W^{*(i)}$ denotes the recovered parameter matrix of the i-th layer, and $W^{(i)}$ denotes the original parameter matrix. Additionally, $w^{(i)}_{m,n}$, where $1 \leq m \leq d_i$ and $1 \leq n \leq d_{i-1}$, represents the (m,n)-th element of $W^{(i)}$. The subsequent discussion focuses on k-layer models.

Permutation of Neurons. During the model extraction attack, although the parameters associated with each neuron in a hidden layer can be recovered, their exact correspondence to a specific row in the parameter matrix remains unknown. For the recovered matrix of first-layer, the row ordering is inconsistent with the original one. For the i-th layer, $1 < i \leq k$, there exist column permutations induced by the row permutations in the $(i-1)$-th layer matrix, in addition to row permutations resulting from the ambiguity in neuron correspondence.

Scaling of Neurons. When extracting the parameters of the u-th neuron in the i-th layer parameter matrix (the u-th row in $W^{(i)}$), its values are scaled by an additional scaling factor, denoted by a constant $c^{(i)}_u$.

For the first-layer parameter matrix, there is only an additional row scaling factor $c^{(1)}_u$, for $1 \leq u \leq d_1$. If row permutations are disregarded, we denote the

(m, n)-th element of the $W^{*(1)}$ by

$$W^{*(1)}[m, n] = c_m^{(1)} w_{m,n}^{(1)}, \quad \text{where } m \in \{1, \ldots, d_1\},\ n \in \{1, \ldots, d_0\}.$$

The parameter matrix $W^{*(i)}$, $1 < i \leq k$, will scale the column values correspondingly, because of the row scaling factor $c_u^{(i-1)}$, for $1 \leq u \leq d_{i-1}$ originating from previous layer $W^{*(i-1)}$. Meanwhile, the u-th row of $W^{*(i)}$ is scaled by an additional neuron scaling factor, $c_u^{(i)}$, for $1 \leq u \leq d_i$. If row permutations and column permutations are not considered, we regard $W^{*(i)}$ and $W^{(i)}$ as aligned. When $W^{*(i)}$ and $W^{(i)}$ are aligned, we consider only the effect of neuron scaling on $W^{*(i)}$. The (m, n)-th element of the matrix is denoted by

$$W^{*(i)}[m, n] = c_m^{(i)} \frac{w_{m,n}^{(i)}}{c_n^{(i-1)}}, \quad \text{where } m \in \{1, \ldots, d_i\},\ n \in \{1, \ldots, d_{i-1}\}.$$

The Recovered Output Layer Parameters. We briefly consider the case where the output layer contains a single neuron. In the S1 scenario, the analysis proposed in [2] indicates that the output layer weight $W^{(k+1)}$ and bias $b^{(k+1)}$ cannot be recovered separately when extracting the parameters of the output layer. Instead, the recovered output layer parameter $W^{*(k+1)}$ is scaled by a value of $\frac{1}{|0.5-b^{(k+1)}|}$, and $b^{*(k+1)} = -1$. The weight of output layer $W^{(k+1)}$ is denoted as $(w_1, \ldots, w_{d_k})$, and the input to the output layer, is denoted as $(v_1, \ldots, v_{d_k})$. The m-th element of the $W^{*(k+1)}$ is denoted by

$$W^{*(k+1)}[m] = \frac{w_m}{c_m^k |0.5 - b^{(k+1)}|}, \quad \text{where } m \in \{1, \ldots, d_k\}.$$

The process of recovering the output layer parameters proposed in [2] is as follows. We first employ the method proposed by [3] to identify dual points. In the case of binary classification, a dual point corresponds to an input for which the output layer score is 0.5. Using the dual points as the model input, the following equation holds:

$$w_1 v_1 + \cdots + w_{d_k} v_{d_k} + b^{(k+1)} = 0.5.$$

Next, the following equation can be solved using more than d_k dual points:

$$\frac{w_1}{0.5 - b^{(k+1)}} v_1 + \cdots + \frac{w_{d_k}}{0.5 - b^{(k+1)}} v_{d_k} = 1.$$

If $b^{(k+1)} < 0.5$, an equivalent network is obtained; if $b^{(k+1)} > 0.5$, the recovered network misclassifies inputs and requires flipping the signs of the recovered output layer parameters. We can compare the outputs of the original and recovered models, and flip the sign if they differ.

2.3 Recurrent Neural Networks

RNN models have been widely applied in language modeling tasks, such as text classification [18], summarization [8], machine translation [10], and image-to-text translation [20], among others. For experiments in this paper, we analyze

the classical RNN architecture proposed in [11]. This RNN model not only revolutionized sequential data processing but also laid the groundwork for subsequent advancements in deep learning, profoundly influencing modern architectures in fields such as natural language processing, speech recognition, and time-series analysis. Subsequent architectures, such as long short-term memory networks [13] and gated recurrent units [6], can generally be considered as variants of the original RNN.

The RNN architecture consists of a basic neural network enhanced with recurrent (feedback) connections. Owing to parameter reusability, it can process sequential data of varying lengths, enabling generalization across sequences. Unlike feedforward neural networks, which assign distinct parameters to each input feature, an RNN model applies the same parameters across multiple time steps. At each time step, the output is computed using the same update rule as in previous steps, making the current output dependent on prior states, while the number of outputs depends on the specific task the RNN is designed to perform. Conceptually, an RNN model can be unfolded into a deep computational graph, in which identical parameters are reused in sequence.

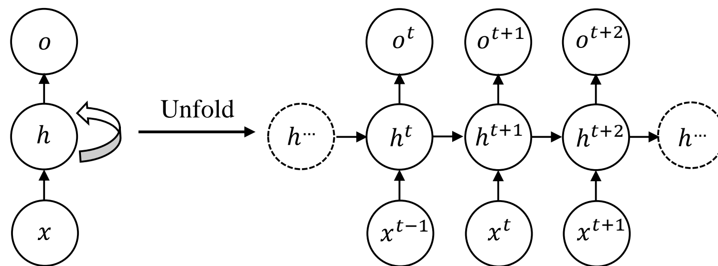


Fig. 2: RNN: (Left) network with feedback connection; (Right) unfolded computational graph across multiple time steps.

An RNN model processes an input sequence in which each value, after embedding, is transformed into a vector x^t. The time step index t ranges from 1 to τ, where τ represents the length of the input sequence (rather than corresponding to real-world time). The unfolded form of the computational graph is shown in Fig. 2. The cycles in the left panel of Fig. 2 capture the influence of previous variable values on the current time step. The right panel of Fig. 2 exhibits a repetitive structure that unfolds the recursive computation of the RNN into a sequence of operations. It depicts the flow of information forward when computing outputs and losses. Let g denote the state transition function of the RNN model. The corresponding update rule is

$$h^{t+1} = g\big(h^t, x^t, W\big), \quad \text{for } 1 \leq t \leq \tau,$$

where h^t denotes the hidden state, x^t the input, t the time step, W the set of network parameters, comprising the input-to-hidden parameters (W_x, b) and the

hidden-to-hidden parameters (W_h, b), which share the same bias b, as well as the hidden-to-output parameters (W_o, b_o). The parameters (W_x, b) map the external input into the hidden representation, (W_h, b) capture temporal dependencies by propagating information across time steps, and (W_o, b_o) transform the hidden state into the final output of the network.

Let x^i (for $1 \leq i \leq \tau$) denote the vector obtained by mapping the i-th token through the embedding layer when RNN models are applied to tasks in natural language processing. d_x denotes the dimensionality of the vector x^i. The number of hidden neurons in the hidden layer of the RNN model is denoted by d_h.

3 Attacks on RNN Models in the S5 Scenario

In this section, we present a complete attack for RNN models in the logit-ouput setting (S5). We begin by introducing the attack scenarios and the adversarial capabilities. Next, we describe the process of converting an RNN model into an equivalent FCN model using model equivalence transformation techniques. To make the extracted parameters reusable, we then propose two methods: a permutation alignment method and a scaling alignment method. Finally, we introduce the recovery of the output layer.

3.1 Attack Scenarios and Adversary Capabilities

The RNN model is among the most widely used neural network models in natural language processing, as its recurrent structure effectively captures sequential dependencies in text.

Attack Scenarios. In our study, the attack scenario is defined as the application of RNN models to binary text classification tasks. Traditionally, text classification tasks are addressed by first splitting the input sequence into tokens, then processing them through an embedding layer to obtain fixed-size vectors. The sequence of vectors is subsequently fed into RNN models that output only the final predicted y, without exposing any intermediate hidden states, as illustrated in Fig. 3.

The parameter dimensions are: $W_x \in \mathbb{R}^{d_h \times d_x}$, $W_h \in \mathbb{R}^{d_h \times d_h}$, $b \in \mathbb{R}^{d_h}$, $W_o \in \mathbb{R}^{d_h}$, and $b_o \in \mathbb{R}$. The initial hidden state $h^1 \in \mathbb{R}^{d_h}$ is set to the zero vector $\mathbf{0}$, and the intermediate states are computed as

$$h^{t+1} = \sigma(W_x x^t + W_h h^t + b).$$

The final output y is computed as

$$y = o^{\tau+1} = W_o h^{\tau+1} + b_o.$$

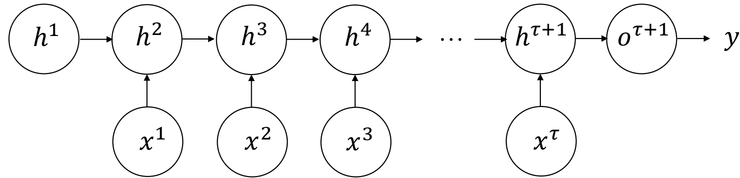


Fig. 3: RNN model for binary text classification.

Adversary Capabilities. We make the following assumptions regarding the target neural network to be extracted (denoted by f) and the attacker's capabilities, as adapted from [1,4] with minor modifications.

- **RNN models with ReLU activation.** f is an RNN model employing the ReLU activation function.
- **Known architecture.** The attacker knows the architecture of f.
- **Unrestricted input access.** The attacker can query the network on any input x^i, for $1 \leq i \leq \tau$, with full control over the inputs after the embedding layer.
- **Raw output access.** f returns the complete raw output, with no post-processing.
- **Precise computations.** f processes inputs using 64-bit arithmetic.

3.2 Idealized Attack Model

To perform a model extraction attack on RNN models, we propose **model equivalence transformation technique** which transforms the RNN model into an FCN model by controlling the input sequence. Once the model equivalence has been established, the corresponding parameter recovery techniques proposed in [1,4,17] can be employed.

To begin with, we can control the input vectors in the simplest way. By fixing $x^i = \mathbf{0}$, for $2 \leq i \leq \tau$ and allowing only a single varying input x^1, the parameter recovery problem of RNN models can be equivalently transformed into the parameter recovery problem of FCN models. After the equivalence transformation, the resulting FCN model has the architecture $d_x - d_h^\tau - 1$.

However, this equivalence has a limitation: according to [4], current end-to-end neural network parameter recovery methods can handle at most 3 layers for an FCN model. When $\tau > 3$, it becomes infeasible to conduct an effective attack using existing techniques.

The above equivalence method cannot be extended to deeper neural networks because it fails to exploit the inherent characteristics of RNN models. Therefore, we propose a novel attack which can recover deeper RNN models, leveraging the intrinsic characteristics of RNN models. The first step of our attack constructs a new equivalent model using the model equivalence transformation method and recovers it. We control the input of the RNN model such that $x^{\tau-2} \neq \mathbf{0}$, whereas all other inputs satisfy $x^t = \mathbf{0}$ for $t \neq \tau - 2$. This setting is illustrated in Fig. 4,

where the green nodes represent zero vectors, the yellow nodes denote nonzero input vectors of the RNN model, and the orange arrows indicate the data flow induced by the nonzero input $x^{\tau-2}$.

The transformation of $x^{\tau-2}$ does not affect the content in the blue box in Fig. 4. The blue box indicates that for $1 \leq t \leq \tau - 3$, $x^t = \mathbf{0}$, which under our controlled input condition yields a fixed vector $h^{\tau-2}$. The initial hidden state is a zero vector, i.e., $h^1 = \mathbf{0}$. The iterative relation for computing $h^{\tau-2}$ is given as follows:

$$h^2 = \sigma(W_h h^1 + b),\ h^3 = \sigma(W_h h^2 + b),\ ...,\ h^{\tau-2} = \sigma(W_h h^{\tau-3} + b).$$

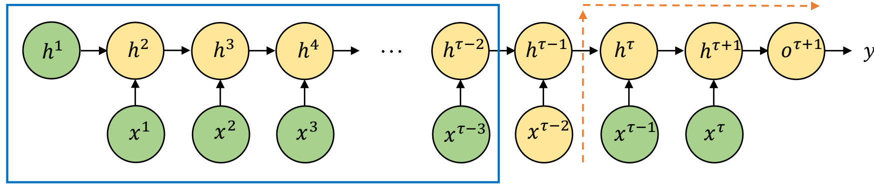


Fig. 4: RNN model input constraint: only $x^{\tau-2}$ is nonzero.

The actually recovered FCN model corresponds to the data flow along the orange arrows in Fig. 4. Based on the data flow, we can compute the intermediate states as:

$$\begin{aligned} h^{\tau-1} &= \sigma(W_x x^{\tau-2} + W_h h^{\tau-2} + b), \quad & h^{\tau} &= \sigma(W_h h^{\tau-1} + b), \\ h^{\tau+1} &= \sigma(W_h h^{\tau} + b), & o^{\tau+1} &= \sigma(W_o h^{\tau+1} + b_o). \end{aligned}$$

Since the blue box generates a fixed bias vector $h^{\tau-2}$, we can define the constant $b_{new} = W_h h^{\tau-2} + b$ for simplicity. Under this controlled input condition, the RNN model can be transformed into an equivalent FCN model with an architecture of dimensions $d_x - d_h^3 - 1$, as illustrated in Fig. 5.

Accordingly, the parameters of the equivalent FCN model, constructed through the model equivalence transformation technique, are expressed as follows:

- **First:** parameter matrix W_x, bias b_{new}.
- **Second:** parameter matrix W_h, bias b.
- **Third:** parameter matrix W_h, bias b.
- **Output:** parameter matrix W_o, bias b_o.

As shown in Fig. 5, the two parts which share identical parameters are highlighted in red, and we additionally recover the third layer matrix, which is identical to the second layer matrix, to facilitate our permutation and scaling alignment method.

Our model equivalence transformation method considers only cases where a single input vector is active under controlled input conditions. The depth of the

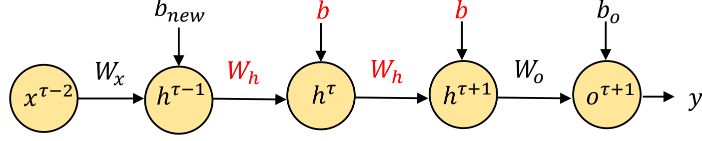


Fig. 5: The equivalent FCN model where only $x^{\tau-2}$ is nonzero.

equivalent FCN model depends on the active position of x under our controlled input condition: if $x^{\tau-1}$ is active, the layer depth is 2; if $x^{\tau-2}$ is active, the layer depth is 3. For the equivalent FCN model obtained by unfolding the RNN model along the data flow, d_x denotes the size of the input layer, while d_h denotes the number of hidden neurons.

In the S5 scenario, the first step of our attack is to recover this equivalent three-layer FCN model with an architecture of $d_x - d_h^3 - 1$, using the model extraction methods proposed in [1,4]. We denote the recovered parameters as $W_x^{*(1)}$, $b_{new}^{*(1)}$, $W_h^{*(2)}$, $b_h^{*(2)}$, $W_h^{*(3)}$, $b_h^{*(3)}$, W_o^*, b_o^*.

It is evident that RNN models have relatively few parameters, with the matrices W_x, W_h, and vector b being reused across time steps. Nevertheless, due to the issues of permutation and scaling of neurons that arise in the recovered parameters, as discussed in Section 2.2, we cannot repeatedly use the previously recovered parameters $W_x^{*(1)}$, $W_h^{*(2)}$, and $b_h^{*(2)}$. Our goal is to adjust the recovered parameters $W_x^{*(1)}$, $W_h^{*(2)}$, and $b_h^{*(2)}$ in order to avoid repeatedly recovering the same parameters. To address this problem, we propose two methods, namely the **permutation alignment method** and the **scaling alignment method**.

In the following, we describe how $W_x^{*(1)}$, $W_h^{*(2)}$, and $b_h^{*(2)}$ are processed to adjust them for reuse. $W_x^{*(1)}$ can be directly reused, whereas $W_h^{*(2)}$ and $b_h^{*(2)}$ require further adjustment. Let Q_i denote the scaling of neurons in the i-th layer. It is a diagonal matrix

$$Q_i = \mathrm{diag}\big(c_1^{(i)}, c_2^{(i)}, \ldots, c_{d_i}^{(i)}\big).$$

Let P_i denote a permutation of the neurons in the i-th layer. We express the recovered parameters in terms of the original parameters (W_x, W_h, W_o, b_{new}, b, b_o) together with the permutation matrices P_i and scaling matrices Q_i:

$$\begin{cases} W_x^{*(1)} = P_1 Q_1 W_x, & b_{new}^{*(1)} = P_1 Q_1 b_{new}, \\ W_h^{*(2)} = P_2 Q_2 W_h Q_1^{-1} P_1^{-1}, & b_h^{*(2)} = P_2 Q_2 b, \\ W_h^{*(3)} = P_3 Q_3 W_h Q_2^{-1} P_2^{-1}, & b_h^{*(3)} = P_3 Q_3 b, \\ W_o^* = W_o Q_3^{-1} P_3^{-1}, & b_o^* = b_o. \end{cases}$$

Let x denote the input to this equivalent FCN model. Using the original paramters, the output of this model y_o is as follows:

$$y_o = W_o \, \sigma\Bigg(W_h \, \sigma\Big(W_h \, \sigma\big(W_x x + b_{new}\big) + b\Big) + b \Bigg) + b_o.$$

In the S5 scenario, we can compute the same output y_o^* as y_o using the recovered parameters. Denote (y_1^*, y_2^*, y_3^*, y_o^*) as the intermediate state values of each layer before ReLU activation. The data flow is as follows, with x as the input to the model with recovered parameters:

$$\begin{aligned}
y_1^* &= P_1 Q_1 W_x x + P_1 Q_1 b_{new}, \\
y_2^* &= (P_2 Q_2 W_h Q_1^{-1} P_1^{-1}) \, \sigma\big(y_1^*\big) + P_2 Q_2 b, \\
y_3^* &= (P_3 Q_3 W_h Q_2^{-1} P_2^{-1}) \, \sigma\big(y_2^*\big) + P_3 Q_3 b, \\
y_o^* &= (W_o Q_3^{-1} P_3^{-1}) \, \sigma\big(y_3^*\big) + b_o \\
&= W_o \, \sigma\Bigg(W_h \, \sigma\Big(W_h \, \sigma\big(W_x x + b_{new}\big) + b\Big) + b \Bigg) + b_o.
\end{aligned}$$

It can also be observed that, although the original parameters in the second and third layers are identical, we cannot directly substitute the recovered second layer parameters for the third layer parameters due to the permutation and scaling of neurons during the actual parameter recovery. If we adjust the recovered parameters of the second layer into the following form,

$$W_h^{*(2)} = P_1 Q_1 W_h Q_1^{-1} P_1^{-1}, \quad b_h^{*(2)} = P_1 Q_1 b,$$

$W_h^{*(2)}$ and $b_h^{*(2)}$ can be directly reused in subsequent layers of the extracted model. When we use the reusable parameters to compute the output for x as the input, their corresponding data flow (y_1', y_2', y_3', y_o') is given as follows:

$$\begin{aligned}
y_1' &= P_1 Q_1 W_x x + P_1 Q_1 b_{new}, \\
y_2' &= (P_1 Q_1 W_h Q_1^{-1} P_1^{-1}) \, \sigma\big(y_1'\big) + P_1 Q_1 b, \\
y_3' &= (P_1 Q_1 W_h Q_1^{-1} P_1^{-1}) \, \sigma\big(y_2'\big) + P_1 Q_1 b. \\
y_o' &= (W_o Q_1^{-1} P_1^{-1}) \, \sigma\big(y_3'\big) + b_o \\
&= W_o \, \sigma\Bigg(W_h \, \sigma\Big(W_h \, \sigma\big(W_x x + b_{new}\big) + b\Big) + b \Bigg) + b_o.
\end{aligned}$$

This data flow clearly indicates that, after adjusting $W_h^{*(2)}$ and $b_h^{*(2)}$ to make the permutation and scaling matrix consistent with the previous layer, the recovered parameters of the second layer can substitute the parameters of the third layer. After adjusting the parameters to make them reusable, the output layer parameters can be recovered as $W_o^* = W_o Q_1^{-1} P_1^{-1}$ and $b_o^* = b_o$.

To make parameters reusable, we first apply the permutation alignment method to compute $P_1P_2^{-1}$. After aligning $W_h^{*(2)}$ and $b_h^{*(2)}$, we obtain

$$W_h^{*(2)} = P_1Q_2W_hQ_1^{-1}P_1^{-1}, \quad b_h^{*(2)} = P_1Q_2b.$$

Next, we apply the scaling alignment method to obtain ratio $= P_1Q_1Q_2^{-1}P_1^{-1}$. Applying this ratio to $W_h^{*(2)}$ and $b_h^{*(2)}$ yields $W_h^{*(2)} = P_1Q_1W_hQ_1^{-1}P_1^{-1}$ and $b_h^{*(2)} = P_1Q_1b$.

3.3 Permutation Alignment Method

If the parameters can be reused, we need to adjust the row permutation of $W_h^{*(2)}$ and $b_h^{*(2)}$ to ensure consistency. The $P_1P_2^{-1}$ matrix is computed from the permutation obtained by aligning $W_h^{*(2)}$ and $b_h^{*(2)}$ based on their sign information.

Since the parameter recovery process preserves the sign of each individual parameter (i.e., whether it is positive or negative), we can align two matrices recovered from the same underlying matrix using their sign information. Specifically, we apply the indicator function $\mathbb{I}(\cdot)$, defined as

$$\mathbb{I}(x > 0) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{otherwise}, \end{cases}$$

to extract the sign information from $W_h^{*(2)}$ and $W_h^{*(3)}$. Denote the resulting sign matrices by $W_h^{*(2)}$ and $W_h^{*(3)}$, respectively. Since $W_h^{*(2)}$ and $W_h^{*(3)}$ are equivalent parameterizations of the same underlying matrix W_h, the two matrices satisfy the following relation:

$$P_2Q_3^{-1}W_h^{*(3)}P_2P_1^{-1} = W_h^{*(2)}.$$

If we can align $W_h^{*(3)}$ and $W_h^{*(2)}$, we can recover the left permutation matrix $P_2Q_3^{-1}$ and the right permutation matrix $P_2P_1^{-1}$.

We introduce a permutation alignment method that exploits row-sum and column-sum information to construct candidate sets for each row and each column. The method iteratively refines these candidate sets by leveraging unique row candidates, i.e., rows in $W_h^{*(2)}$ that admit a single candidate, to prune column candidates and, symmetrically, by using unique column candidates to prune row candidates, as shown in Algorithm 1. Since the pruning process may generate new unique row and column candidates, we iteratively apply the pruning procedure until no new unique row or column candidates are produced in an iteration.

The probability that at least one unique row candidate exists during the alignment process is equivalent to the probability that at least one row sum appears exactly once in an $n \times n$ matrix after applying the indicator function. Next, we provide a theoretical approximation of this probability.

Algorithm 1 Permutation Alignment Method

1: **Input:** $W_s^{*(2)}, W_s^{*(3)}$
2: **Output:** $\mathcal{R}_{cand}$, $\mathcal{C}$ ▷ $\mathcal{C}_{cand}$: set of column indices
3: SumRow(M), SumCol(M) $\rightarrow$ row/column sums of M
4: **function** GenCand($\mathbf{c}_x$, $\mathbf{c}_h$) ▷ Returns candidate index sets
5: Initialize empty list $\mathcal{W}$
6: **for** each $c_x^{(i)} \in \mathbf{c}_x$ **do**
7: $\mathcal{I} \leftarrow \{j \mid \mathbf{c}_h^{(j)} = c_x^{(i)}\}$
8: Append $\mathcal{I}$ to $\mathcal{W}$
9: **return** $\mathcal{W}$
10: **function** FindUnique($\mathcal{W}$)
11: Initialize empty list $\mathcal{P}$
12: **for** each $(i, \mathcal{I}_i) \in$ enumerate($\mathcal{W}$) **do**
13: **if** $|\mathcal{I}_i| = 1$ **then**
14: Append $(i,\ \mathcal{I}_i[0],\ 0)$ to $\mathcal{P}$
15: **return** $\mathcal{P}$
16: $\mathbf{s}_r^{(k)} \leftarrow \text{SumRow}(W_s^{*(k)}),\ \mathbf{s}_c^{(k)} \leftarrow \text{SumCol}(W_s^{*(k)}),\quad k = 2, 3$
17: $\mathcal{R}_{\text{cand}} \leftarrow \text{GenCand}(\text{s}_r^{(2)},\ \text{s}_r^{(3)}),\ \mathcal{C}_{\text{cand}} \leftarrow \text{GenCand}(\text{s}_c^{(2)},\ \text{s}_c^{(3)})$
18: $\mathcal{P}_r \leftarrow \text{FindUnique}(R_{\text{cand}}),\ \mathcal{P}_c \leftarrow \text{FindUnique}(C_{\text{cand}})$
19: **if** $|\mathcal{P}_r| = 0$ **and** $|\mathcal{P}_c| = 0$ **then**
20: $(\mathcal{R}_{cand}, \mathcal{C}_{cand}) \leftarrow$ FilterCandTwo($\mathcal{R}_{cand}, \mathcal{C}_{cand}$)
21: **while true do**
22: $|\mathcal{P}_c|_{\text{old}} \leftarrow |\mathcal{P}_c|$
23: **for** each $(i,\ \hat{i},\ \delta) \in \mathcal{P}_r$ **do**
24: **if** $\delta = 1$ **then**
25: **continue**
26: **for** $j \leftarrow 0$ **to** ncol$(W_s^{*(3)}) - 1$ **do**
27: $v \leftarrow W_s^{*(2)}[i,\ j]$
28: $\mathcal{C}_{\text{cand}}[j] \leftarrow \{\ \hat{j} \in C_{\text{cand}}[j] \mid W_s^{*(3)}[\hat{i},\ \hat{j}] = v\ \}$
29: Mark $(i, \hat{i}, \delta)$ as used: $\delta \leftarrow 1$
30: $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{(j, \hat{j}, 0) \mid (j, \hat{j}) \in \text{FindUnique}(\mathcal{C}_{\text{cand}}) \wedge (j, \hat{j}) \notin \mathcal{P}_c\}$
31: $|\mathcal{P}_c|_{\text{new}} \leftarrow |\mathcal{P}_c|$
32: $|\mathcal{P}_r|_{\text{old}} \leftarrow |\mathcal{P}_r|$
33: **for** each $(j,\ \hat{j},\ f) \in \mathcal{P}_c$ **do**
34: **if** $f = 1$ **then**
35: **continue**
36: **for** $i \leftarrow 0$ **to** nrow$(W_s^{*(3)}) - 1$ **do**
37: $v \leftarrow W_s^{*(2)}[i,\ j]$
38: $\mathcal{R}_{\text{cand}}[i] \leftarrow \{\ \hat{i} \in \mathcal{R}_{\text{cand}}[i] \mid W_s^{*(3)}[\hat{i},\ \hat{j}] = v\ \}$
39: Mark $(j, \hat{j}, \delta)$ as used: $\delta \leftarrow 1$
40: $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{(i, \hat{i}, 0) \mid (i, \hat{i}) \in \text{FindUnique}(\mathcal{R}_{\text{cand}}) \wedge (i, \hat{i}) \notin \mathcal{P}_r\}$
41: $|\mathcal{P}_r|_{\text{new}} \leftarrow |\mathcal{P}_r|$
42: **if** $|\mathcal{P}_r|_{\text{new}} = |\mathcal{P}_r|_{\text{old}}\ \wedge\ |\mathcal{P}_c|_{\text{new}} = |\mathcal{P}_c|_{\text{old}}$ **then**
43: **break**
44: **return** $\mathcal{R}_{\text{cand}}$, $\mathcal{C}_{\text{cand}}$

Assume that M is an $n \times n$ matrix whose entries are independent Bernoulli random variables, taking the value 0 with probability 1/2 and the value 1 with probability 1/2, where 0 represents a negative sign and 1 represents a positive sign. Let S_i denote the sum of the i-th row. For each $k \in \{0, \ldots, n\}$, define the event

$$A_k := \{\text{the value } k \text{ appears exactly once among } S_1, \ldots, S_n\}.$$

We further define the event

$$U := \bigcup_{k=0}^{n} A_k,$$

which corresponds to the existence of at least one unique row sum.

Then

$$p_k = \Pr(S_i = k) = \binom{n}{k} 2^{-n},$$

and the probability of the event A_k is given by

$$\Pr(A_k) = n\, p_k (1 - p_k)^{n-1}.$$

Summing over all possible values of k yields

$$\lambda = \sum_{k=0}^{n} \Pr(A_k),$$

which represents the expected number of row sums that occur exactly once. Under a Poisson approximation, we have

$$\Pr(U) \approx 1 - e^{-\lambda}.$$

In practice, since both unique row candidates and unique column candidates can be exploited during the alignment process, the probability that a row or a column admits a unique candidate can be expressed as

$$1 - e^{-2\lambda}, \tag{1}$$

which increases with n, reaching 99.759% when $n = 25$ and a value whose deviation from 1 is on the order of 10^{-11} when $n = 1000$.

When n is relatively small and no unique row or column candidate can be identified, we propose Algorithm 6 to reduce row or column candidate lists of size two to a unique candidate, as described in Appendix A.1. The core idea of Algorithm 6 is to traverse candidate lists of size two and eliminate candidates that do not satisfy the required consistency conditions. For example, consider a row candidate for which there exists a column such that the corresponding value in $W_s^{*(2)}$ does not match any of the values associated with the corresponding column candidates in $W_s^{*(3)}$; in this case, the row candidate is filtered out. During the execution of Algorithm 1, if no unique candidate is available, Algorithm 6 is executed.

Table 2: Results of the permutation alignment method.

Dimensions	Num_{no}	Time
25×25	7	3.5×10^{-3}s
1000×1000	0	4.13s

To evaluate the effectiveness of our permutation alignment method, we conduct the following experiments. In each experiment, we generate an $n \times n$ matrix M_1, where each element independently takes the value 0 or 1 with probability 1/2. We then generate a random $n \times n$ row permutation matrix P and a random $n \times n$ column permutation matrix Q, and compute

$$M_2 = PM_1Q.$$

The goal of the experiment is to verify whether our permutation alignment algorithm can recover the permutation matrices P and Q given M_1 and M_2. For matrices of size 25×25 and 1000×1000, we perform 10,000 permutation alignment trials. As summarized in Table 2, our method successfully aligns the matrices in all experiments.N_{no} denotes the number of trials (out of 10,000) in which no unique candidate can be identified based on the sums of row and column signs. We observe that for the 25×25 alignment task, the absence of unique candidates occurs with an extremely small probability, which is consistent with our theoretical analysis. In such cases, by combining Algorithm 6 with Algorithm 1, we can still successfully complete the alignment. The column *Time* reports the runtime required for completing the alignment.

Algorithm 1 takes $W_s^{*(2)}$ and $W_s^{*(3)}$ as inputs and returns the permutation matrices $\mathcal{R}(P_2Q_3^{-1})$ and $\mathcal{C}(P_2P_1^{-1})$. The algorithm first computes the row sums and column sums of $W_s^{*(2)}$ and constructs, for each row and each column, a candidate set of possible matches in $W_s^{*(3)}$.

Next, the algorithm identifies unique row candidates, namely rows in $W_s^{*(2)}$ whose candidate sets contain exactly one element. Once such a row index r is fixed, the algorithm iterates over all column indices to perform pruning. For a given column index c, the entry $W_s^{*(2)}[r, c]$ is compared with the corresponding entries $W_s^{*(3)}[r_{\text{cand}}, c_{\text{cand}}]$, where r_{cand} denotes the uniquely determined matching row in $W_s^{*(3)}$, and c_{cand} ranges over the column candidates associated with column c. If the two values are inconsistent, the corresponding column candidate is pruned. An analogous pruning procedure is applied using unique column candidates to further eliminate row candidates. Since each pruning step may generate new unique row or column candidates, the procedure is applied iteratively until no new unique candidates are produced.

The Algorithm 6 takes $\mathcal{R}_{\text{cand}}, \mathcal{C}_{\text{cand}}, W_s^{*(2)}$, and $W_s^{*(3)}$ as inputs. The algorithm traverses candidate lists whose size is exactly two and filters invalid candidates based on consistency checks. For example, consider a row index r whose candidate list contains two candidates. We temporarily fix one row can-

didate r_{cand} and traverse all column candidate lists. For each column index c, we examine all associated column candidates c_{cand}. If the corresponding entry $W_s^{*(2)}[r,c]$ does not match any of the entries $W_s^{*(3)}[r_{\text{cand}}, c_{\text{cand}}]$, then the fixed row candidate r_{cand} is eliminated. In this case, the remaining candidate becomes a unique row candidate. The filtering procedure for column candidates is carried out analogously.

Complexity The worst-case time complexity of Algorithm 6 for aligning an $n \times n$ matrix is $O(n^3)$. In the worst case, the algorithm processes at most $2n$ row and column candidates whose candidate lists have size two. For each such candidate value, the procedure traverses up to n rows or columns, and each checking step involves comparing up to n matrix entries.

Therefore, the overall worst-case time complexity is

$$4n \times n \times n = O(n^3).$$

The worst-case time complexity of Algorithm 1 for aligning an $n \times n$ matrix is $O(n^3)$. In the worst case, the algorithm processes at most $2n$ unique candidates, including both unique row candidates and unique column candidates. For each unique candidate, the pruning procedure may traverse up to n row or column candidates, and each pruning step involves comparing up to n matrix entries. Therefore, taking into account the complexity of Algorithm 6, the overall worst-case time complexity is

$$2n \times n \times n + 4n \times n \times n = O(n^3).$$

3.4 Scaling Alignment Method

In the preceding section, we introduced a method for achieving permutation alignment; we now address the issue of neuron scaling, with the objective of enforcing consistency in the row scaling, namely $c_u^{(2)} = c_u^{(1)}$ for $1 \le u \le d_h$.

As discussed in Section 3.2, the second and third layers share the same matrix W_h and bias b. We would like to exploit this property to compute the ratio

$$\frac{c_u^{(1)}}{c_u^{(2)}}, \quad \text{for } 1 \le u \le d_h.$$

First, based on the fact that the biases of the second and third layers are equal, we can ensure that $c_u^{(2)} = c_u^{(3)}$, for $1 \le u \le d_h$, by normalizing the biases during the parameter recovery process, which facilitates the computation of the ratio. In order to justify this equality, we analyze the relationship between the recovered biases $b_h^{*(2)}$ and $b_h^{*(3)}$ and the true bias b, disregarding permutation effects. From the discussion in Section 3.3, we know that the additional scaling factors $c_u^{(1)}$, for $1 \le u \le d_h$, in the first layer only affect the column scaling of $W_h^{*(2)}$, and do

not influence the scaling of $b_h^{*(2)}$. Therefore, only the row scaling of the second layer affects the recovered bias $b_h^{*(2)}$. Since $b_h^{*(2)}$ is a vector, we can denote

$$b_h^{*(2)}[i] = c_i^{(2)}\, b[i], \quad \text{for } 1 \leq i \leq d_h.$$

The same argument applies to the third layer

$$b_h^{*(3)}[i] = c_i^{(3)}\, b[i], \quad \text{for } 1 \leq i \leq d_h.$$

Following parameter recovery techniques [1,4], the recovered biases $b_h^{*(2)}$ and $b_h^{*(3)}$ in the practical recovery procedure can be normalized to ± 1 by simultaneously adjusting their values together with the corresponding rows of the parameter matrices, without affecting the overall parameter recovery. Since the biases of the second layer and third layer parameters are both denoted by b (with consistent signs after normalization), their normalized values satisfy

$$b_h^{*(2)}[i] = b_h^{*(3)}[i], \quad \text{for } 1 \leq i \leq d_h.$$

And the following is directly obtained:

$$c_i^{(2)} = c_i^{(3)}, \quad \text{for } 1 \leq i \leq d_h.$$

We conclude that if the original biases in the second and third layers are identical, the biases can be normalized while simultaneously adjusting the corresponding parameter matrices row-wise, so that the recovered scaling of the second and third layers remains consistent. In Section 2.2, without considering permutations, we denote by $w_{m,n}$, for all the entries of the matrix W_h. So the entries of $W_h^{*(2)}$ are given by

$$W_h^{*(2)}[m,n] = c_m^{(2)} \frac{w_{m,n}}{c_n^{(1)}}, \quad \text{where}\, m,n \in \{1,\ldots,d_h\}.$$

Similarly, for the third layer, the entries of the matrix $W_h^{*(3)}$ are

$$W_h^{*(3)}[m,n] = c_m^{(3)} \frac{w_{m,n}}{c_n^{(2)}}, \quad \text{where}\, m,n \in \{1,\ldots,d_h\}.$$

Under the condition that $c_i^{(2)} = c_i^{(3)}$, for $1 \leq i \leq d_h$, it follows that $\frac{W_h^{*(3)}}{W_h^{*(2)}}[m,n] = \frac{c_n^{(1)}}{c_n^{(2)}}$, for $1 \leqslant m,\ n \leqslant d_h$. By selecting any row from $\frac{W_h^{*(3)}}{W_h^{*(2)}}$, e.g., the 0-th row, we directly obtain the scaling vector $\frac{c_i^{(1)}}{c_i^{(2)}}$, where $1 \leqslant i \leqslant d_h$.

Applying the inverse of $\mathcal{C}$ to $W_h^{*(2)}$ resolves the neuron permutation ambiguity and transforms $W_h^{*(2)} = P_2 Q_2 W_h Q_1^{-1} P_1^{-1}$ into $W_h^{*(2)} = P_1 Q_2 W_h Q_1^{-1} P_1^{-1}$. The same operation also transforms $b_h^{*(2)} = P_2 Q_2 b$ into $b_h^{*(2)} = P_1 Q_2 b$.

Subsequently, by applying the computed ratio to $W_h^{*(2)}$, we obtain

$$W_h^{*(2)} = P_1 Q_1 W_h Q_1^{-1} P_1^{-1}.$$

Similarly, $b_h^{*(2)} = P_1 Q_1 b$ is obtained by an analogous transformation.

3.5 Recovery of the Output Layer

After applying the permutation and scaling alignment method, we proceed to recover the output layer parameters W_o^* and b_o^*. The parameters of the last layer are recovered via the least squares method, along with the substitution of the original parameters by the adjusted parameters $W_x^{*(1)}$, $W_h^{*(2)}$, and $b_h^{*(2)}$, as illustrated in Fig. 6. The parameters to be recovered are indicated by the purple box. Let $h^{(\tau+1)}$ denote the output of the τ-th layer in the original model. Using the substituted parameters, the τ-th layer output is computed as $h^{*(\tau+1)} = P_1 Q_1 h^{\tau+1}$. Consequently, the recovered output-layer parameters satisfy $W_o^* = W_o Q_1^{-1} P_1^{-1}$ and $b_o^* = b_o$, where

$$h^{\tau+1} = W_h\, \sigma\Big(W_h\, \sigma\big(W_x x + b_{new}\big) + b\Big) + b,$$

$$h^{*(\tau+1)} = P_1 Q_1 W_h\, \sigma\Big(W_h\, \sigma\big(W_x x + b_{new}\big) + b\Big) + P_1 Q_1 b.$$

Therefore, given M distinct input vectors with $M > d_h$, we are able to compute the corresponding inputs and outputs of the final layer. As a result, the parameters of the final layer, W_o^* and b_o^*, can be recovered using the least squares method.

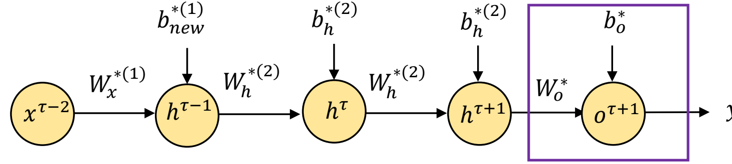


Fig. 6: Recovery of the Final Layer Parameters.

In the following, we provide a brief introduction to the complete attack pipeline for RNN models, as shown in Algorithm 2. First, we apply the model equivalence transformation method to construct an equivalent FCN model with a $d_x - d_h^3 - 1$ architecture. The model is then extracted using the approach proposed by [4], with the bias values normalized to ± 1 in the process. Second, we employ the permutation and scaling alignment method to enable the reuse of the intermediate-layer parameter. Finally, based on the previously constructed equivalent FCN model with a $d_x - d_h^3 - 1$ architecture and incorporating the reusable parameters, the output layer parameters (W_o^* and b_o^*) are recovered using the least squares method. The implementation details of the complete attack pipeline are presented in Algorithm 2.

4 Attacks on RNN Models in the S1 Scenario

This section introduces the implementation of a complete end-to-end attack on RNN models in the S1 scenario. If we can recover the three-layer FCN model

Algorithm 2 RNN Attack Pipeline in the S5 Scenario

1: **Input:** An RNN model
2: **Output:** the functionally equivalent RNN model $\left(W_x^{*(1)},\, W_h^{*(2)}, b_h^{*(2)}, W_o^*, b_o^*\right)$
3: $W_x^{*(1)}, W_h^{*(2)}, b_h^{*(2)}, W_h^{*(3)}, b_h^{*(3)} \leftarrow$ Parameter extraction with $x^t = \mathbf{0},$ for $t \neq \tau - 2$
4: $\mathcal{R}, \mathcal{C} \leftarrow$ Permutation Alignment Method$(W_h^{*(2)}, W_h^{*(3)})$
5: $W_h^{*(3)} \leftarrow \mathcal{R} W_h^{*(3)} \mathcal{C}$
6: ratio $\leftarrow \frac{W_h^{*(3)}}{W_h^{*(2)}}[0,\ :]$
7: Initialize W as an empty matrix and b as an empty vector
8: **for** $i = 0$ **to** $d_h - 1$ **do**
9: $\quad W[i, :] \leftarrow W_h^{*(2)}[\mathcal{C}[i],\ :]$
10: $\quad b \leftarrow b_h^{*(2)}[\mathcal{C}[i]]$
11: **for** $j = 0$ **to** $d_h - 1$ **do**
12: $\quad W_h^{*(2)}[j,\ :] \leftarrow \text{ratio}[j] \cdot W[j,\ :]$ ▷ Each row is rescaled by a ratio
13: $\quad b_h^{*(2)}[j] \leftarrow \text{ratio}[j] \cdot b[j]$
14: $W_o^*, b_o^* \leftarrow$ Least-Squares Estimation based on $W_x^{*(1)}, W_h^{*(2)}, b_h^{*(2)}$
15: **return** $W_x^{*(1)},\ W_h^{*(2)},\ b_h^{*(2)},\ W_o^*,\ b_o^*$

in the S1 scenario, the attack methods for the S5 scenario are also applicable. However, achieving a complete end-to-end attack is more challenging in the S1 scenario. Therefore, we propose a new equivalence transformation for RNN models that reduces the recovery problem to that of two FCN models with a maximum depth of two. To enable parameter reuse, we propose a permutation search strategy that adjusts the permutation of intermediate-layer parameters and calibrates the scaling factors based on the recovered output-layer parameters of the two models. Furthermore, to address the challenges of recovering the parameters of equivalent FCN models in the S1 scenario, we propose an accuracy enhancement method and a more robust sign search method.

4.1 Complete Attack Pipeline and Supporting Methods

The attack scenario and the adversary's capabilities in the S1 scenario are essentially the same as those in the S5 scenario, as described in Section 3.1, except that the adversary only has access to the model's predicted label rather than the raw output of the RNN model.

To enable a complete end-to-end attack, we propose a new equivalent transformation for an RNN model, which requires the recovery of only a two-layer FCN model and a one-layer FCN model.

The first step of our attack pipeline is to recover the parameters of two equivalent FCN models in the S1 scenario. The one-layer FCN model is obtained by fixing the value of $x^\tau \neq \mathbf{0}$ and enforcing $x^i = \mathbf{0}$ for all $1 \leq i \leq \tau - 1$, as illustrated in Fig. 7. The yellow nodes indicate that the corresponding values are non-zero vectors, while the green nodes denote zero vectors.

The orange arrows represent the data flow in the equivalent model. Based on this data flow, we can identify the corresponding parameters of the equivalent

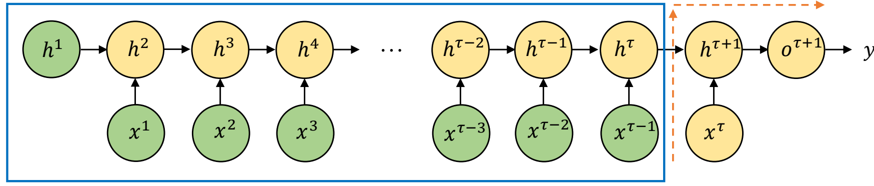


Fig. 7: RNN model input constraint: only x^τ is nonzero.

FCN model, as illustrated in the left diagram of Fig. 8, where $b_\tau = W_h h^\tau + b$. The iterative relation for computing h^τ is given as follows:

$$h^2 = \sigma(W_h h^1 + b),\ \ldots,\ h^{\tau-1} = \sigma(W_h h^{\tau-2} + b),\ h^\tau = \sigma(W_h h^{\tau-1} + b).$$

The two-layer equivalent FCN model is obtained by fixing the value of $x^{\tau-1}$ and enforcing the constraint $x^i = \mathbf{0}$ for all $i \neq \tau - 1$, as illustrated in Fig. 9. The parameters of the equivalent model can be found in the right diagram of Fig. 8, where $b_{\tau-1} = W_h h^{\tau-1} + b$.

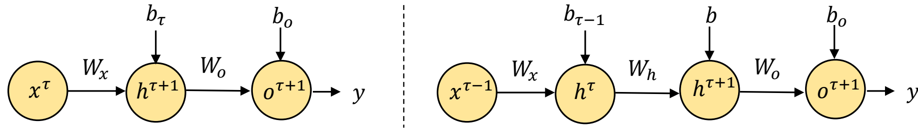


Fig. 8: (Left) Equivalent FCN model with only x^τ nonzero; (Right) Equivalent FCN model with only $x^{\tau-1}$ nonzero.

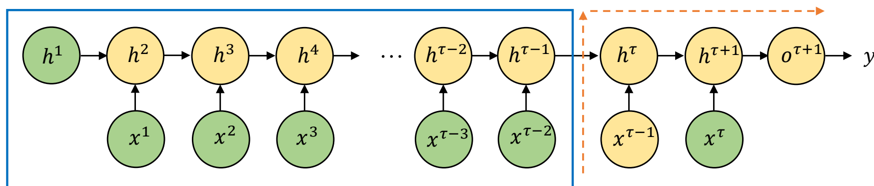


Fig. 9: RNN model input constraint: only $x^{\tau-1}$ is nonzero.

As discussed in Section 2.2, we can represent the recovered equivalent model using the permutation and scaling transformation matrices along with the original parameters. For the sake of notational simplicity, we focus on the case where the output-layer bias satisfies $b_o < 0.5$. In this setting, the recovered bias is fixed to -1, and the output weight W_o is rescaled as $W_o \leftarrow \frac{W_o}{0.5 - b_o}$.

In the S1 scenario, the parameters of the one-layer equivalent recovered FCN model are denoted as

$$W'_o = W_o Q_1^{-1} P_1^{-1}, \quad W'_x = P_1 Q_1 W_x, \quad b'_o = \mathbf{0}, \quad b'_x = P_1 Q_1 b_\tau.$$

Because the first-layer weights of the two-layer FCN model are identical to those of the one-layer FCN model, once the first-layer weights of the one-layer FCN model are recovered, they can be directly reused. The corresponding bias can then be recovered using the dual-point technique. The parameters of the second recovered two-layer FCN model are given by

$$\begin{aligned} W''_o &= W_o Q_2^{-1} P_2^{-1}, & W''_h &= P_2 Q_2 W_h Q_1^{-1} P_1^{-1}, & W''_x &= P_1 Q_1 W_x, \\ b''_o &= -\mathbf{1}, & b''_h &= P_2 Q_2 b, & b''_x &= P_1 Q_1 b_{\tau-1}. \end{aligned}$$

After recovering two equivalent FCN models in the S1 scenario, the second step aims to make the recovered parameters reusable. Specifically, we transform the hidden-layer parameters as

$$W''_h \leftarrow P_1 Q_1 W_h Q_1^{-1} P_1^{-1}, \qquad b''_h \leftarrow P_1 Q_1 b.$$

To make W''_h and b''_h reusable, we propose a **permutation search strategy** to adjust them from the perspective of rows, as detailed in Algorithm 3. Once the permutation matrix $P_1 P_2^{-1}$ is determined, the corresponding scaling transformation matrix $P_1 Q_1 Q_2^{-1} P_1^{-1}$ can be derived from W'_o and W''_o. If the guessed permutation matrix is correct, W''_h and b''_h can be reused, allowing the recovery of an equivalent model. Specifically, we compute

$$W'_o (P_1 P_2^{-1}) = W_o Q_2^{-1} P_1^{-1},$$

and then take the element-wise ratios between $W_o Q_2^{-1} P_1^{-1}$ and the corresponding entries of W'_o. These ratios allow us to recover the scaling transformation matrix $P_1 Q_1 Q_2^{-1} P_1^{-1}$.

The permutation search strategy guesses the permutation and leverages two sign-based constraints to effectively reduce the search space. The first constraint exploits the sign consistency between $W'_o(P_1 P_2^{-1}) = W_o Q_1^{-1} P_2^{-1}$ and $W''_o = W_o Q_2^{-1} P_2^{-1}$. Since the permutation and scaling of neurons do not affect the parameter signs, the correct permutation $P_1 P_2^{-1}$ must satisfy the condition of sign consistency between $W_o Q_1^{-1} P_2^{-1}$ and $W_o Q_2^{-1} P_2^{-1}$.

The second constraint exploits the sign consistency between $b'_x = P_1 Q_1 b_\tau$ and the computed value $(P_1 P_2^{-1})\Big(W''_h\,\sigma(b''_x) + b''_h\Big)$. Since b is a vector and $b_\tau = W_h\,\sigma(b_{\tau-1}) + b$, the computation can be expressed as

$$\begin{aligned} (P_1 P_2^{-1})\Big(W''_h\,\sigma(b''_x) + b''_h\Big) &= (P_1 P_2^{-1})\Big(P_2 Q_2 W_h Q_1^{-1} P_1^{-1}\,\sigma(P_1 Q_1 b_{\tau-1}) + P_2 Q_2 b\Big) \\ &= P_1 Q_2 b_\tau. \end{aligned}$$

Therefore, the search space can be reduced by exploiting the sign consistency between the computed values $P_1 Q_2 b_\tau$ and $P_1 Q_1 b_\tau$.

In the reduced space, we search for $P_1 P_2^{-1}$, and its correctness is verified through the dual points, as outlined in Algorithm 3. The procedure for validating a guessed permutation is as follows. First, a three-layer equivalent FCN model is constructed based on model equivalence transformation method, by controlling input with $x^{\tau-2} \neq \mathbf{0}$ and $x^t = \mathbf{0}$ for all $t \neq \tau - 2$.

Then, we can find the dual points of this three-layer equivalent model. The bias for the first layer of this equivalent model, $P_1 Q_1 b_{\tau-2}$, can be computed based on our adjusted parameters, while the parameters of the other layers can be applied directly. Therefore, we can use the recovered parameters to build a functionally equivalent FCN model, and assess the correctness of the permutation based on the output of the dual points.

Algorithm 3 Permutation Search Strategy

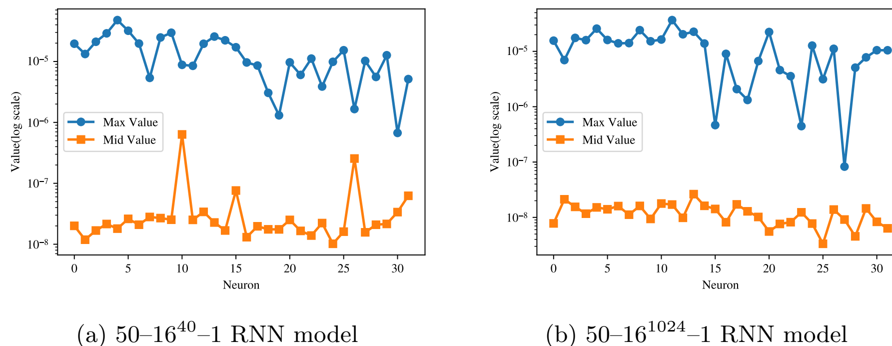
1: **Input:** W_x'', b_x'', W_h'', b_h'', W_o', b_o', W_o'', dual_points
2: **Output:** W_h'', b_h''
3: **function** matmul(W, X, b)
4:     **return** $WX + b$
5: **function** FindCond(W_1, W_2) ▷ Return conditions for the permutation
6:     $l^+ \leftarrow \{j \mid W_2[j] > 0\}, \quad l^- \leftarrow \{j \mid W_2[j] < 0\}$
7:     **return** $[\, l^+$ if $W_1[i] > 0$ else $l^- \mid i = 0, \ldots, \dim(W_1) - 1\,]$
8: **function** RunDualPoints(W_x'', b_{new}, W, b, W_o', b_o') ▷ Validate the permutation
9:     $\mathcal{D} \leftarrow$ dual_points
10:     **for** (w, b) in $\big([W_x'',\ W,\ W],\ [b_{new},\ b,\ b]\big)$ **do**
11:         $\mathcal{D} \leftarrow \sigma(\text{matmul}(\mathcal{D}, w, b))$
12:     $\mathcal{D} \leftarrow \text{matmul}(\mathcal{D}, W_o', b_o')$
13:     **if** $\mathcal{D} = 1$ with tolerance 10^{-6} **then return** true
14:     **else** **return** false
15: $c_x'' \leftarrow W_h'' \, \sigma(b_x'') + b_h''$
16: cond_1 $\leftarrow$ FindCond(W_o', W_o''), cond_2 $\leftarrow$ FindCond(b_x', c_x'')
17: $\text{list}_{set}[i] \leftarrow \text{cond_1}[i] \cap \text{cond_2}[i], \quad \forall i \in \{0, \ldots, d_h - 1\}$
18: Partition $\{\text{list}_{set}[i]\}_{i=0}^{d_h-1}$ into $\{V_1, \ldots, V_k\}$, Define $\mathcal{P} = \prod_{j=1}^{k} \text{Perm}(V_j)$
19: **for** all $p \in \mathcal{P}$ **do**
20:     Initialize $W \leftarrow$ zero matrix (shape W_h'') and $b \leftarrow$ zero vector (shape b_h'')
21:     **for** $i = 0$ to $\text{len}(p) - 1$ **do**
22:         ratio $\leftarrow \frac{W_o''[p[i]]}{W_o'[i]}$
23:         $W[i] \leftarrow W_h''[p[i],\ :] \cdot \text{ratio}$ ▷ Each row is rescaled by a ratio
24:         $b[i] \leftarrow b_h''[p[i]] \cdot \text{ratio}$
25:     $h_t \leftarrow \mathbf{0}$
26:     **for** $j = 0$ to $\tau - 4$ **do**
27:         $h_t \leftarrow \sigma(\text{matmul}(h_t, W, b))$
28:     $b_{new} \leftarrow \text{matmul}(h_t, W, b)$
29:     **if** RunDualPoints(W_x'', b_{new}, W, b, W_o', b_o') **then**
30:         $W_h'' \leftarrow W$, $b_h'' \leftarrow b$
31:         **return** W_h'', b_h''

4.2 Improvements on Parameter Recovery Techniques

Accuracy Enhancement Method. For dual points used in parameter recovery, the corresponding intermediate neuron values in the neural network should be sufficiently close to zero, as this property has a significant impact on the accuracy of parameter recovery.

Compared to the S5 scenario, the S1 scenario is substantially more challenging. In particular, we cannot determine whether the derivative of the output changes in order to precisely identify critical inputs. Instead, critical planes are detected by traversing along the input space and observing whether the plane exhibits curvature. As a result, numerical inaccuracies accumulated during this process may cause some dual points to deviate from ideal critical points, rendering them insufficiently accurate in the end-to-end attack setting.

For the two RNN models, we separately examine the neuron state values corresponding to the dual points within the dual-point clusters used for neuron parameter recovery. The results are shown in Fig. 10. For the equivalent FCN models corresponding to two different RNN models, we separately evaluate, for each cluster, the curve of the maximum neuron state value (shown in blue) and the curve of the median state value (shown in orange). Neurons indexed from 0 to 15 correspond to the first layer, while indices from 16 to 31 correspond to the second layer. We observe that some dual points are not sufficiently accurate. In particular, their distances from zero are noticeably larger compared to those of other points. If dual points that are not sufficiently accurate are used during parameter recovery, the accuracy of the recovered parameters may be degraded.



(a) 50–16^{40}–1 RNN model (b) 50–16^{1024}–1 RNN model

Fig. 10: Neuron State Values Corresponding to Dual Points

To filter out insufficiently accurate points, we propose an algorithm for improving recovery accuracy, as shown in Algorithm 4. The core idea of our algorithm is to discard dual points that exhibit large variance.

The function GenSubForward in Algorithm 4 is responsible for generating subspace points and performing the corresponding forward propagation. The

function RecoverW recovers the weight parameters via singular value decomposition (SVD), while RecoverB computes the bias parameters based on the recovered weights. Finally, the predicate IsHits is used to verify whether, for each neuron in the layer preceding the target layer, there exists at least one dual point whose corresponding neuron state is active.

Algorithm 4 takes as input a list of acceptable variance thresholds, denoted by T_{list}, which is sorted in ascending order (e.g., $[10^{-16}, 10^{-15}, \ldots]$) to guarantee that the algorithm eventually returns a result. Once a variance threshold is selected, the algorithm attempts to recover the weight vector w using the current set of dual points P_{temp}. For each dual point in P_{temp} and the corresponding generated subspace point, denoted by P_{matrix}, the variance of the projected neuron state $P_{\text{matrix}}w$ is computed.

Our key observation is that dual points derived from insufficiently accurate dual points tend to induce larger and more unstable fluctuations in the neuron state values $P_{\text{matrix}}w$. Therefore, by filtering out dual points with larger variance, the recovered weight parameters can achieve higher accuracy. If the variances associated with all dual points are below the selected threshold, the algorithm proceeds to recover the bias parameter and returns both the weight and bias.

Otherwise, to improve efficiency, the algorithm discards the half of dual points with the largest variances. Each candidate set for removal, denoted by D_{will}, is further validated by checking whether the remaining dual points still activate all neurons in the layer preceding the target layer. If $D_{\text{will}} \neq \emptyset$, the algorithm updates P_{temp} by removing D_{will} and repeats the above procedure.

If $D_{\text{will}} = \emptyset$, the filtering process cannot proceed under the current configuration, as no dual point can be safely removed. In this case, the algorithm adds P_{temp} to D_{all}. Without changing the current variance threshold, the algorithm then removes D_{all} from the original input set P_{set} and continues the filtering process on the remaining dual points. For intermediate-layer parameters, the condition $D_{\text{will}} = \emptyset$ is used as the criterion for adding P_{temp} to D_{all}. In contrast, for the first layer, the corresponding criterion is $|P_{\text{temp}}| = 2$.

When removing D_{all} from P_{set} leaves an empty set, or when the remaining dual points no longer activate all neurons in the layer preceding the target layer, the algorithm concludes that the weight vector w cannot be recovered under the current threshold. It then proceeds to a larger variance threshold to relax the constraint.

Sign Search Method. To perform an end-to-end attack, we introduce a new sign search method specifically designed for the penultimate layer, as presented in Algorithm 5. We guess the parameters of the penultimate layer and exploit the recovery of the final layer to confirm that the sign recovery of the penultimate layer has been performed correctly. If the guessed signs are correct, the entire reconstructed model is guaranteed to be correct. We then employ dual points to verify whether the recovered equivalent model yields an output of 1, which serves as a criterion for assessing the correctness of our sign recovery.

Algorithm 5 returns parameters of the penultimate-layer with the correct signs as well as the output layer. For ease of exposition, Algorithm 5 targets

Algorithm 4 Accuracy Enhancement Method

```
1:  Input: P_set, T_list
2:  Output: w, b
3:  for each τ ∈ T_list do
4:      D_all ← ∅
5:      while true do
6:          P_temp ← P_set \ D_all
7:          if |P_temp| = 0 then
8:              break
9:          if IsHits(P_temp) = false then
10:             break
11:         while true do
12:             P_all ← GenSubForward(P_temp)
13:             w ← Recover_W(P_all)
14:             V ← ∅
15:             for each P_matrix ∈ P_all do
16:                 V ← V ∪ {Var(P_matrix · w)}
17:             if ∀ v ∈ V, v < τ then
18:                 b ← Recover_B(P_all)
19:                 return (w, b)
20:             I_del ← arg top_⌈(|P_temp|+1)/2⌉(V)
21:             D_will ← { p_i ∈ P_temp | i ∈ I_del }
22:             D_will ← { p ∈ D_will | IsHits(P_temp \ {p}) = true }
23:             if |D_will| = 0 then
24:                 D_all ← D_all ∪ P_temp
25:                 break
26:             P_temp ← P_temp \ D_will
```

the recovery of the signs associated with W_h'' and b_h'' in the equivalent model $RNN_{\tau-1}$. In Algorithm 5, the function Solve_Last_Layer denotes the procedure used to recover the parameters of the output layer, following the method proposed by [2]. The function Run_Model produces the outputs of the dual points using the recovered parameters.

Algorithm 5 Sign Search Method

1: **Input:** W_x'', b_x'', W_h'', b_h'', dual_points
2: **Output:** W_h'', b_h'', W_o''
3: Construct sign_combs as the set of all 2^{d_h} sign vectors in $\{\pm\}^{d_h}$.
4: **for** each sign_t $\in$ sign_combs **do**
5: Initialize $W \leftarrow$ Zeros(shape(W_h''))
6: Initialize $b \leftarrow$ Zeros(shape(b_x''))
7: **for** $i = 0$ to $d_h - 1$ **do**
8: $W[i, :] \leftarrow$ sign_t$[i] \cdot W_h''[i,\ :]$ ▷ Each row is rescaled by a ratio
9: $b[i] \leftarrow$ sign_t$[i] \cdot b_h''[i]$
10: $W_o'' \leftarrow$ Solve_Last_Layer(dual_points, W_x'', b_x'', W_h, b_h)
11: Out $\leftarrow$ Run_Model(dual_points, W_x'', b_x'', W_h, b_h, W_o'')
12: **if** Out is element-wise within 10^{-6} tolerance of 1 **then**
13: $W_h'' \leftarrow W$, $b_h'' \leftarrow b$
14: **return** W_h'', b_h'', W_o''

5 Experiments

In this section, we introduce the training pipeline for RNN models, and the evaluation metrics for recovered RNN models.

RNN Training Pipeline. For the training of RNN model, we employ the IMDB dataset [19], accessed via the Keras library [7], which consists of 25,000 training and 25,000 testing samples annotated with binary sentiment labels. To reduce computational complexity, the vocabulary is restricted to the 5,000 most frequent words. Each sequence is truncated or padded to a fixed length of τ tokens, where τ denotes the depth at which the RNN model is reformulated into an FCN model. Word embeddings are obtained using Word2Vec [21] with the skip-gram algorithm on the IMDB corpus.

For the IMDB dataset, 80% of the training samples (20000 samples) are used as the training set, while the remaining 20% (5000 samples) form the validation set; the test set corresponds directly to the provided testing samples (25000 samples). Training was performed for 40 epochs, with all datasets processed in mini-batches of size 32. Models were trained using the binary cross-entropy loss and the Adam optimizer [15] with the default Keras [7] settings. The learning rate was fixed at 0.001 throughout training.

Evaluation Metrics and Results. The aim of the extraction is not to replicate the target network exactly, but rather to obtain what is referred to as an (ϵ, δ)-functionally equivalent extraction [4]. This metric is calculated by comparing the output of the target model with that of the recovered equivalent model, both of which take the same input. Each input sample of the RNN models consists of τ embedding vectors, denoted as s. The L samples are randomly generated, which together constitute the overall input space, denoted as S. The definition of (ϵ, δ) is thus given as follows.

Definition 4. *Two models f and g are (ϵ, δ)-functionally equivalent on S if*

$$\Pr_{s \in S} [\, |f(s) - g(s)| \leq \epsilon \,] \geq 1 - \delta.$$

We adopt the (ϵ, δ) evaluation metric as the basis for model assessment and set $L = 2 \times 10^6$. To facilitate clarity, the RNN models are represented using their equivalent FCN model formulations. For example, consider an RNN model with embedding dimension $d_x = 50$, hidden size $d_h = 16$, and sequence length $\tau = 40$ (each sample contains τ embedding vectors). The equivalent structure is denoted as $50 - 16^{40} - 1$.

In the S5 scenario, we provide five attack results of RNN models with different configurations, including $50-16^{1024}-1$, $50-20^{40}-1$, $50-16^{100}-1$, $100-16^{40}-1$, $800-16^{40}-1$. Our end-to-end attack results provide the deepest attack results. In addition, our attack methods are theoretically not limited by network depth. In the S1 scenario, we adopt the same evaluation metrics (ϵ, δ) to evaluate the recovered RNN model. However, since the recovered parameters of the output layer contain a scaling factor $1/|0.5 - b_o|$, we remove this factor before evaluating our recovered RNN model. This operation is used to facilitate evaluation and has no impact on our attack results. We present the attack results for the $50-16^{40}-1$ model, which represents the deepest model extraction attacks in the S1 scenario.

6 Conclusion

In this paper, we introduced the first cryptanalytic extraction attack for RNN models in both the S1 and S5 scenarios. By exploiting the inherent parameter-reuse property of RNN models, our attack enables the extraction of much deeper models compared to previous publications. Our end-to-end attack extracts RNN models of up to 1024 layers in the S5 scenario and 40 layersl in the S1 scenario, respectively. The main steps of the attack are similar in the S1 and S5 scenarios, converting the RNN model into FCN models via the model equivalence transformation method, and then adjusting the parameters for reuse. In the S5 scenario, we convert the recovered RNN parameters into a three-layer FCN model. To address the problem of neuron permutation and scaling in making the parameters reusable, we propose the permutation and scaling alignment method. Since extracting an end-to-end three-layer FCN in the S1 scenario is challenging with current methods, we introduce an alternative approach for constructing equivalent models. The recovered RNN parameters are transformed

into two FCN models of depth at most two. To make parameters reusable, we propose a value-based alignment method and a permutation search strategy. In addition, to address the challenges encountered during the model extraction attack in the S1 scenario, we propose a sign search method for recovering the signs of the penultimate layer and a filter method to facilitate clustering.

References

1. Canales-Martínez, I.A., Chávez-Saab, J., Hambitzer, A., Rodríguez-Henríquez, F., Satpute, N., Shamir, A.: Polynomial time cryptanalytic extraction of neural network models. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT 2024. pp. 3–33. Springer Nature Switzerland, Cham (2024)
2. Canales-Martínez, I.A., Santos, D.: Extracting some layers of deep neural networks in the hard-label setting. Cryptology ePrint Archive (2025)
3. Carlini, N., Chávez-Saab, J., Hambitzer, A., Rodríguez-Henríquez, F., Shamir, A.: Polynomial time cryptanalytic extraction of deep neural networks in the hard-label setting. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025. pp. 364–396. Springer Nature Switzerland, Cham (2025)
4. Carlini, N., Jagielski, M., Mironov, I.: Cryptanalytic extraction of neural network models. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020. pp. 189–218. Springer International Publishing, Cham (2020)
5. Chen, Y., Dong, X., Guo, J., Shen, Y., Wang, A., Wang, X.: Hard-label cryptanalytic extraction of neural network models. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024. pp. 207–236. Springer Nature Singapore, Singapore (2025)
6. Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: Encoder-decoder approaches. In: Wu, D., Carpuat, M., Carreras, X., Vecchi, E.M. (eds.) Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014. pp. 103–111. Association for Computational Linguistics (2014). https://doi.org/10.3115/V1/W14-4012, https://aclanthology.org/W14-4012/
7. Chollet, F., et al.: Keras. https://keras.io (2015)
8. Chopra, S., Auli, M., Rush, A.M.: Abstractive sentence summarization with attentive recurrent neural networks. In: Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. pp. 93–98 (2016)
9. Coqueret, B., Carbone, M., Sentieys, O., Zaid, G.: A hard-label cryptanalytic extraction of non-fully connected deep neural networks using side-channel attacks. CoRR **abs/2411.10174** (2024). https://doi.org/10.48550/ARXIV.2411.10174, https://doi.org/10.48550/arXiv.2411.10174
10. Datta, D., David, P.E., Mittal, D., Jain, A.: Neural machine translation using recurrent neural network. International Journal of Engineering and Advanced Technology **9**(4), 1395–1400 (2020)
11. Elman, J.L.: Finding structure in time. Cogn. Sci. **14**(2), 179–211 (1990), https://doi.org/10.1207/s15516709cog1402_1
12. Foerster, H., Mullins, R.D., Shumailov, I., Hayes, J.: Beyond slow signs in high-fidelity model extraction. In: Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J.M., Zhang, C. (eds.) Advances in Neural

Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024 (2024), http://papers.nips.cc/paper_files/paper/2024/hash/22ae669a35bb9e70eb93ab77c1eff5b4-Abstract-Conference.html
13. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/NECO.1997.9.8.1735, https://doi.org/10.1162/neco.1997.9.8.1735
14. Ito, A., Miura, T., Todo, Y.: Is the hard-label cryptanalytic model extraction really polynomial? arXiv preprint arXiv:2510.06692 (2025)
15. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980
16. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998). https://doi.org/10.1109/5.726791
17. Liu, H., Siproudhis, A., Experton, S., Lorenz, P., Boura, C., Peyrin, T.: Navigating the deep: Signature extraction on deep neural networks. arXiv preprint arXiv:2506.17047 (2025)
18. Liu, P., Qiu, X., Huang, X.: Recurrent neural network for text classification with multi-task learning. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016. pp. 2873–2879. IJCAI/AAAI Press (2016), http://www.ijcai.org/Abstract/16/408
19. Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C.: Learning word vectors for sentiment analysis. In: Lin, D., Matsumoto, Y., Mihalcea, R. (eds.) The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA. pp. 142–150. The Association for Computer Linguistics (2011), https://aclanthology.org/P11-1015/
20. Mao, J., Xu, W., Yang, Y., Wang, J., Yuille, A.L.: Deep captioning with multimodal recurrent neural networks (m-rnn). In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6632
21. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: Bengio, Y., LeCun, Y. (eds.) 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013), http://arxiv.org/abs/1301.3781
22. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. In: Fürnkranz, J., Joachims, T. (eds.) Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel. pp. 807–814. Omnipress (2010), https://icml.cc/Conferences/2010/papers/432.pdf
23. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review **65**(6), 386 (1958)
24. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by backpropagating errors. nature **323**(6088), 533–536 (1986)
25. Takemura, T., Yanai, N., Fujiwara, T.: Model extraction attacks against recurrent neural networks. CoRR **abs/2002.00123** (2020), https://arxiv.org/abs/2002.00123

A Appendix

A.1 Filter Candidate Two Method

Algorithm 6 Filter Candidate Two Method

```
1: Input: R_cand, C_cand, W_s^{*(2)}, W_s^{*(3)}
2: Output: R_cand, C_cand
3: function FindTwo(W)
4:     Initialize empty list P
5:     for each (i, I_i) ∈ enumerate(W) do
6:         if |I_i| = 2 then
7:             Append (i, I_i[0,1], 0) to P
8:     return P
9: P_r ← FindTwo(R_cand)
10: P_c ← FindTwo(C_cand)
11: for each (r_i, R_val) ∈ P_r do
12:     f_filter ← false
13:     for each r_g ∈ R_val do
14:         for j = 0 to cols(W_s^{*(2)}) − 1 do
15:             v_r ← W_s^{*(2)}[r_i][j]
16:             f_satisfy ← false
17:             for each c ∈ C_cand[j] do
18:                 v_g ← W_s^{*(3)}[r_g][c]
19:                 if v_r = v_g then
20:                     f_satisfy ← true
21:                     break
22:             if f_satisfy then
23:                 continue
24:             else
25:                 f_filter ← true
26:                 break
27:         if f_filter then
28:             R_cand ← R_cand \ {r_g}
29:             break
30: for each (c_i, C_val) ∈ P_c do
31:     f_filter ← false
32:     for each c_g ∈ C_val do
33:         for i = 0 to rows(W_s^{*(2)}) − 1 do
34:             v_r ← W_s^{*(2)}[i][c_i]
35:             f_satisfy ← false
36:             for each c ∈ R_cand[i] do
37:                 v_g ← W_s^{*(3)}[c][c_g]
38:                 if v_r = v_g then
39:                     f_satisfy ← true
40:                     break
41:             if f_satisfy then
42:                 continue
43:             else
44:                 f_filter ← true
45:                 break
46:         if f_filter then
47:             C_cand ← C_cand \ {c_g}
48:             break
    return R_cand, C_cand
```