NeuPerm: Disrupting Malware Hidden in **Neu**ral Network Parameters by Leveraging **Per**mutation Symmetry

Daniel Gilkarov
Department of Computer and Software Engineering
Ariel Cyber Innovation Center
Ariel University, Israel
daniel.gilkarov1@msmail.ariel.ac.il
Ran Dubin
Department of Computer and Software Engineering
Ariel Cyber Innovation Center
Ariel University, Israel
rand@ariel.ac.il

***Abstract*—Pretrained deep learning model sharing holds tremendous value for researchers and enterprises alike. It allows them to apply deep learning by fine-tuning models at a fraction of the cost of training a brand-new model. However, model sharing exposes end-users to cyber threats that leverage the models for malicious purposes. Attackers can use model sharing by hiding self-executing malware inside neural network parameters and then distributing them for unsuspecting users to unknowingly directly execute them, or indirectly as a dependency in another software. In this work, we propose NeuPerm, a simple yet effective way of disrupting such malware by leveraging the theoretical property of neural network permutation symmetry. Our method has little to no effect on model performance at all, and we empirically show it successfully disrupts state-of-the-art attacks that were only previously addressed using quantization, a highly complex process. NeuPerm is shown to work on LLMs, a feat that no other previous similar works have achieved. The source code is available at https://github.com/danigil/NeuPerm.git.**

I. Introduction

Deep Learning (DL) neural network models are a major technological advancement that has been used and applied extensively in various fields: Computer Vision (CV) [39], [8], [4], Natural Language Processing (NLP) [30], [5], [12], [42], in particular, the topic of Large Language Models (LLM) is drawing enormous attention [46], [38], [29], and many more [33], [19], [27], [26]. Neural network models, while very flexible and capable, require vast resources [18] such as expertise, computation power, data, etc., to train models from scratch on a given task. LLMs in particular are very useful, but they require a meticulous and resource-heavy training process. For example, training Meta's Llama1 65B LLM took 2048 A100 GPUs with 80GB of RAM, and the training process took approximately 21 days [37]. The alternative is to use Pre-Trained Models (PTM) shared online on PTM hubs such as HuggingFace [17], and TensorFlow-Hub [36]. The PTMs, when used correctly, serve as an enormous shortcut to applying a neural network model to some task [18]. The necessity and value of PTM sharing are clear. However, PTM sharing introduces the potential for ML supply chain attacks, for example, attacks where attackers upload models infected with malware to the PTM sharing hubs [24], [40], [41], [15], [14]. The topic of ML supply chain attacks has been listed as a threat to ML security by different projects that map cyber threats: MITRE ATLAS (Adversarial Threat Landscape for Artificial-Intelligence Systems) [28] indicates ML supply chain compromise as one of the threat vectors in Artificial-Intelligence (AI) systems, the OWASP top 10 for LLM applications 2025 also does so [31]. Securing the ML supply chain is paramount to ensuring end-user safety and continuous, unhindered collaboration on ML and DL.

Past research demonstrates how attackers can use data-hiding (steganography) techniques to hide malware inside neural network parameters. Song et al. [34] initially introduced Least Significant Bit (LSB) substitution, sign-mapping (or sign-encoding), and correlated value encoding neural network steganography techniques that they used for data exfiltration attacks. StegoNet [24], a subsequent work, formulated the neural network steganography attack and researched 4 neural network steganography techniques - LSB substitution, sign-mapping, value-mapping, and resilience training. They design 2 trigger mechanisms, show the methods successfully evade detection by MetaDefender and traditional steganalysis and evaluate model accuracy before and after the attack, which causes up to 2% test accuracy drops to CNNs in most cases. EvilModel [40], [41] improved on StegoNet, they focused on LSB substitution and aimed to reduce the performance drop the steganography causes. They claim almost half of the CNN models used could be attacked with up to 48.52% Embedding Rate (ER) with no performance hit as opposed to only up to 15% ER with StegoNet. MaleficNet [15], [14] improved the previous neural network steganography works by focusing on increasing evasiveness, improving resilience to corruption of the payload by using error-correcting codes,

and improving the trigger mechanism, which extracts and executes the malware upon loading of the model, instead of triggering based on some condition as in EvilModel [40], [41]. After having a payload embedded in their parameters using neural network steganography, the resulting models are stegomalwares (malicious software that uses steganographic techniques) that can potentially cause damage. These stegomalwares pose a security threat to the ML supply chain, as they can be easily propagated, do not degrade model performance [41], [15], [14], and are shown to successfully avoid detection by common anti-virus software [14], malware detection systems, and steganalysis methods [45]. Moreover, as opposed to steganography carried out in traditional digital media types such as images [35], audio [2], or text [25], neural networks have a much greater capacity for hidden data, since they are much larger files. For illustration, JPG images usually weigh around 3 megabytes, and in contrast, neural network models sometimes weigh multiple gigabytes; the Llama3.3 70B LLM weighs approximately 140 gigabytes.

In this work, we focus on mitigating these neural network model stegomalwares. Past efforts to combat neural network stegomalwares took 2 approaches: **(1)**: zero-trust disruption - these works aim to damage a potential embedded payload by changing the parameters; adding random noise to the LSBs, and fine-tuning [6], [3] was shown to work on naive LSB substitution neural network steganography like StegoNet and EvilModel [24], [40], [41], but not on methods with error-correction such as MaleficNet [15], [14]; another method is compressing the weights using quantization [6], which seems to work even for the resilient error-correcting steganography introduced in MaleficNet [15], [14], however, it lowers model performance [6] and post-training quantization is highly specific [21]. **(2)**: steganography detection (steganalysis) - these works aim to create techniques to classify samples as "innocent" (no payload) or "malicious", similar to steganalysis works in other domains [20]. Zhao et al. [45] suggested steganalysis methods aimed to detect the methods introduced by Song et al. [34], which were proposed for data exfiltration attacks; while this is useful for these attacks, they might not generalize to unseen attacks like value-mapping(StegoNet) or MaleficNet [24], [15], [14]. Gilkarov et al. [11], [10] trained supervised classification models to detect LSB substitution steganography with an ER of up to 25%, and they show trained methods generalize well to unseen attacks such as MaleficNet; however, the training process is complex, and it is fitted to a certain attack. In this work we present **NeuPerm**, a simple, yet effective zero-trust technique for disrupting neural network steganography that leverages the permutation symmetry of neural networks [13]. See Fig. 1, our method has almost no impact on model performance, it is simpler compared with model compression, and through evaluation, we conclude that it can disrupt resilient techniques like MaleficNet, which as far as we know, is a first in the academic literature.

The key contributions of our research are enumerated below:

- We introduce **NeuPerm**: A simple, yet effective zero-trust technique for disrupting neural network steganography.
  - The first work to cover LLMs, and the first to disrupt the MaleficNet [15] state-of-the-art attack.
  - Almost no performance drop in the models (upwards of a magnitude of 0.01% score drop).
  - Can be easily extended to uncovered neural network types.
  - Generic method: independent of model size, parameter types, task, structure, etc.
- We conduct a comprehensive evaluation of NeuPerm in terms of effect on model performance and effectiveness that includes different CNNs and an LLM, various malware payloads, 3 validation datasets, and comparison with a widely-cited baseline.
- We publish our source code at https://github.com/danigil/NeuPerm.git.

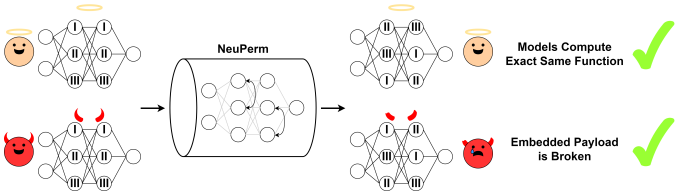

Fig. 1. Illustration of **NeuPerm**. We permute the hidden units of the hidden layers of the neural network. The resulting models compute the same function as they did before the transformation. If the parameters contain an embedded payload, the attackers will not be able to extract it.

II. Threat Model

In this section, we present our threat model. Our threat model is similar to the one presented in past works such as StegoNet [24], and MaleficNet [15].

A. System Entities

The **End-User (Model Consumer)** may be any organization that ingests externally supplied model files, often using pre-trained models from PTM hubs. They wish to leverage PTMs, mainly caring about the model performance.

The **Adversary** is an untrusted third-party that supplies malicious model files embedding executable or exfiltration payloads via steganographic techniques such as those demonstrated by StegoNet [24] or MaleficNet [15]. The adversary aims to launch an attack on the end-user via their malicious models, be it by causing RCE on the end-user's machine or by covertly delivering malicious payloads.

B. Adversary Capabilities and Assumptions

Inspired by the StegoNet threat model [24, §3], we adopt the following capabilities:

1) **Supply-chain insertion.** The adversary can publish or otherwise deliver a tampered model artifact to the consumer.
2) **Model-only control.** The attacker may arbitrarily modify *static* contents of the model file (topology metadata, tensor values, ...) but *cannot* directly execute code on the consumer's host prior to model loading.

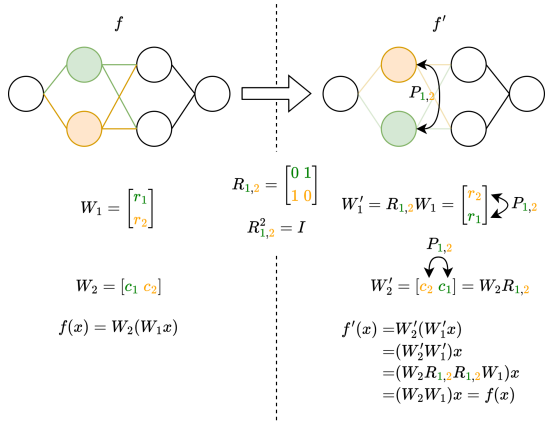

Fig. 2. Multi-Layer Perceptron (MLP) parameter permutation symmetry. Swapping units in a hidden layer does not change the computation; however, it does change the parameter matrices.

3) **No direct network channel.** Once the model is deployed within the consumer's isolated environment, the adversary has no interactive connectivity to the host.
4) **Trigger selection.** The attacker can embed a *logic bomb* that activates on specific inputs (e.g., logits-rank triggers) or upon deserialisation of the model (similar to MaleficNet [15]).

III. Neural Network Parameter Permutation

Neural network architectures often exhibit a property called parameter permutation symmetry [13], this property holds whenever neurons inside a hidden layer of the neural network can be permuted and the neural network still computes the same function. This can be expressed informally as:

$$\forall x, f(x) = f'(x)$$

where f is some feedforward neural network, and f' is the same neural network with some of its hidden layer units shuffled. Past works thoroughly covered parameter symmetry, initially for MLPs, and channel permutation of CNNs [13], [7], [1], and later generalized to any arbitrary feedforward neural network architecture [22]. See Section A for our proofs of the parameter permutation symmetry of select architectures.

A. NeuPerm Implementation

We proceed to describe how we apply NeuPerm in practice based on the proven parameter permutation symmetry properties [13], [22], [7]. Let us define some terms for convenience. We use abbreviations to refer to commonly used neural network components: MLP/Fully Connected (FC), CNN/CONV (can be a 1D or 2D convolutional block), Batch Normalization (BN), Layer Normalization (LN), Average/Max Pooling (AVGP/MAXP), Embedding (EMB). In the following sections, we express a sequential series of neural network blocks like so: BLOCK1-BLOCK2-$\cdots$; for example, CONV-BN-CONV means a convolutional block, followed by batch normalization, and finally, another convolutional block.

1) FC-FC: See Fig. 2. For a FC-FC block, with parameter matrices $W1, b1, W2, b2$ (the first layer has n units) we perform: Given PI some permutation of the indexes $[0, 1, ..., n]$;

```
W1,b1 = W1[PI], b1[PI]
W2 = W2[:,PI]
```

2) CONV-CONV/CONV-BN-CONV: For CONV blocks, we focus on square filter Conv2D blocks to simplify the explanation. DL libraries like PyTorch/TensorFlow initialize a weight matrix of the shape (n, m, k, k) for Conv2D blocks with n $k \times k$ filters with m channels. So, to apply NeuPerm, given permutation indexes PI, we permute the first axis of the first CONV's parameter matrices (weights and biases), and the second axis of the second CONV's parameter matrix, very similarly to the **FC-FC** case. In Python code, it looks like so:

```
W1,b1 = W1[PI], b1[PI]
W2 = W2[:,PI,...]
```

If there is an affine BN layer between the CONVs, we also permute the first axis of its parameter matrices.

3) ATTN: For ATTN blocks, we permute the heads. When writing this article, modern LLMs usually use Grouped Query Attention (GQA), hence, we adapt our method to it. Given an ATTN block with embedding size d, h_q query heads, h_{kv} key-value heads, and G groups: The query, key, value, and projection parameter matrices will be $W_q \in \mathbb{R}^{h_{kv} \times G \times d}$, $W_o \in \mathbb{R}^{d \times \times G \times h_{kv}}$, $W_k, W_v \in \mathbb{R}^{h_{kv} \times d}$. Let PI, permutation indexes of size h_{kv}. In Python code, it looks like so:

```
W_q, W_o = W_q[PI], W_o[..., PI]
W_k, W_v = W_k[PI], W_v[PI]
```

B. NeuPerm on Selected Architectures

In this section, we detail how we apply NeuPerm to specific, well-known state-of-the-art neural network architectures. Section III-A describes how we apply NeuPerm to individual neural network layers, and now we specify how we choose to apply it to a full neural network. We dub the strategies described here as "full NeuPerm". For evaluation, we also use "partial" versions of these strategies to analyze how effectiveness changes as NeuPerm permutes an increasing fraction of the model.

1) Well-Known CNNs: In this section, we detail how we apply NeuPerm to well-known CNNs. We use DenseNet, ResNet, and VGG, and we feel this sufficiently covers the usual CNN architectures. Based on the theory [13], [22], [7], we can apply neural permutations in the following commonly used neural network blocks:

1) FC-FC
2) CONV-CONV
3) CONV-BN-CONV

This is enough to cover a significant portion of CNN architectures. Note: We have to take special care with residual connections, as computations where there is a residual connection in the middle of a sequence like CONV-CONV do not have the parameter permutation symmetry.

DenseNet121: DenseNet121 is structured like so: CONV-MAXP-DB1-T1-DB2-T2-DB3-T3-DB4-CLS. Dense Blocks

TABLE I
SUMMARY OF FRACTION OF PARAMETERS CHANGED BY CHOSEN NEUPERM STRATEGY FOR SELECT CNNS.

Model	#Parameters	#Changed Parameters	Fraction of Changed Parameters
DenseNet121	8M	6.1M	76%
ResNet50	25M	20M	80%
ResNet101	44.5M	39.6M	88%
VGG11	132.9M	132.9M	100%

(DB) have nested Dense Layers (DL) with a CONV-BN-CONV sequence. DLs within DBs are interconnected residually, which leads us to focus on just permuting the inner CONV-BN-CONV sequences within DLs. Transition (T) blocks only have a CONV-AVGP sequence, which is not suited for permutation. Likewise, the Classification (CLS) block is just a GLOBP-FC sequence.

ResNet50/101: ResNet50 and ResNet101 are structured like so: CONV-MAXP-RB1-RB2-RB3-RB4-AVGP-FC. They have a similar structure to the DenseNet121 network. Residual Blocks (RB) have nested Residual Layers (RL) with a CONV-BN-CONV-BN-CONV sequence. RLs within RBs are interconnected residually. Similar to DenseNet121, we choose to permute the inner CONV-BN-CONV-BN-CONV sequences.

VGG11: The VGG family of CNNs comprises just CONV-CONV sequences and two FC-FC sequences, without residual connections. We permute all of these sequences.

See Table I for a summary of the percentage of parameters changed using the described strategies.

2) Well-Known LLMs: In this section, we detail how we apply NeuPerm to transformer-based LLMs. We use the Llama-3.2-1B model in our experiments; however, the described methodology can be easily applied to other LLMs.

Llama-3.2-1B: The Llama-3.2-1B model is structured like so: EMB-LB1-LB2-$\cdots$-LB16-FC. Llama Blocks (LB) contain the sequence LN-ATTN-LN-MLP. We permute the ATTN block and the MLP block of all LBs. This results in $889M/1.5B \approx 60\%$ parameters changed.

IV. EFFECTIVENESS OF NEUPERM

In this section, we discuss the scenarios where NeuPerm can be effective. Most neural network steganography attacks detailed in past works suffer from the same general weakness: extracting the embedded payload requires the parameters of the stegomodel to remain unchanged. This weakness is directly indicated by the authors of neural network steganography attack papers such as StegoNet [24], EvilModel [40], [41]. Due to this, immediate countermeasures such as adding noise, fine-tuning, or pruning are effective against these methods. However, MaleficNet [15], [14] directly addresses this weakness by designing the attack with error-correcting capability. They empirically demonstrate that the malware embedded in MaleficNet stegomodels can withstand adding noise to parameters, fine-tuning, and parameter pruning. Quantization is indicated as a potentially effective counter-measure to MaleficNet, but they argue end-users are unlikely to use such techniques that require expertise and high effort.

Using NeuPerm on applicable models is clearly effective on the fragile steganography techniques (see Section V-D), such as sign-mapping, value-mapping, and LSB substitution (Song et al. [34], StegoNet [24], EvilModel [40]). These techniques rely on the stegomodel staying exactly the same, and by reordering the parameters, NeuPerm changes the model. Hence, **we focus on empirically analyzing NeuPerm on the MaleficNet models**.

V. FORMAL SECURITY ANALYSIS

In this section we analyze the effect of NeuPerm's layer-wise random permutations on an attacker's ability to recover a steganographic payload. We model each permutable layer as drawing an independent uniform permutation on its output axis, then bound the attacker's success probability under this abstraction. Finally, we remark on the obvious limitation that an attacker may choose non-permutable architectures to evade NeuPerm entirely.

A. Model and Notation

Let f_θ be a neural network with parameter set θ, composed of m layers $\mathcal{L} = \{1, \ldots, m\}$. Let

$$\mathcal{L}_{\text{perm}} \subseteq \mathcal{L}$$

be the subset of layers that admit permutation symmetry (e.g. fully-connected, convolutional, or attention blocks). For each $l \in \mathcal{L}_{\text{perm}}$, denote:

- n_l: the *output dimension* of layer l (number of units or convolutional channels).
- $\Theta^{(l)} \in \mathbb{R}^{n_l \times d_l}$: the weight matrix or reshaped tensor whose first axis of length n_l is permutable.
- S_{n_l}: the symmetric group on n_l elements.

Note: For simplicity, we assume each layer has one weight matrix, and that the adversary only embeds 1 bit in each parameter. Let L_{TOTAL} be the number of bits that can be embedded in the model in total, L_{NP} the number of bits that NeuPerm can permute, and L' the number of bits the adversary wishes to embed in the model. Whenever $L' \leq (L_{TOTAL} - L_{NP})$, the adversary can embed the payload only in places that NeuPerm doesn't affect, and thus, NeuPerm is ineffective in these cases. We therefore go forward assuming $L' > (L_{TOTAL} - L_{NP})$, and we set $L := L' - (L_{TOTAL} - L_{NP})$, the amount of bits that must be embedded in places where NeuPerm can act. NeuPerm operates by sampling, *independently* for each $l \in \mathcal{L}_{\text{perm}}$,

$$P_l \xleftarrow{\$} \text{Uniform}\big(S_{n_l}\big),$$

and applying it to the first axis of $\Theta^{(l)}$. Equivalently, if θ collects all parameters, define

$$\mathcal{G} = \prod_{l \in \mathcal{L}_{\text{perm}}} S_{n_l}, \qquad P = (P_l)_{l \in \mathcal{L}_{\text{perm}}} \in \mathcal{G},$$

and let $\widehat{\theta} = P(\theta)$ be the permuted parameters. By construction $f_\theta = f_{\widehat{\theta}}$.

B. Security Game

We adopt the standard extraction game:

1. Adversary $\mathcal{A}$ chooses a clean model θ and payload $m \in \{0,1\}^{L'}$.
2. $\mathcal{A}$ embeds m via a steganographic encoder: $\theta^\star = \mathsf{Enc}(\theta, m)$.
3. Defender samples $P \sim \mathrm{Uniform}(\mathcal{G})$ and publishes $\widehat{\theta} = P(\theta^\star)$.
4. $\mathcal{A}$ receives $\widehat{\theta}$ (and black-box access to $f_{\widehat{\theta}}$), then outputs $\widetilde{m}$.

The adversary's extraction success is

$$S \;:=\; \Pr\big[\widetilde{m} = m\big].$$

C. Fixed-Point Probability per Bit

An embedding oblivious to P places each payload bit in some weight coordinate (l, i, j) with $l \in \mathcal{L}_{\mathrm{perm}}$ and $i \in \{1, \ldots, n_l\}$. Under a uniform P_l, the event that the bit's row-index i remains unchanged has probability $1/n_l$. Hence:

Lemma V.1. *Let $X_k \in \{0,1\}$ indicate that the kth embedded bit survives permutation (remains in its original coordinate). If that bit lies in layer l, then*

$$\Pr[X_k = 1] \;=\; \frac{1}{n_l} \;\le\; d \quad \textit{where} \quad d \;:=\; \max_{l \in \mathcal{L}_{\mathrm{perm}}} \frac{1}{n_l}.$$

Moreover, across bits embedded in distinct weight positions these indicators are independent.

Proof. Uniform randomness of P_l gives $\Pr[P_l(i) = i] = 1/n_l$. Independence follows from the independent draws P_l across layers, and from disjoint embedding positions within each layer. □

D. Attacks With No Error-Correction

We begin by first analyzing the adversary's success probability S whenever the attack used has no error correction capabilities (i.e., Song et al. [34], StegoNet [24], EvilModel [40]). Under this model of attack, we can bound S like so:

$$S = \Pr[X_1 = 1 \wedge X_2 = 1 \wedge \cdots \wedge X_L = 1] \le d^L$$

Where the first transition is due to the X_k's being independent. Now, it is clear that for any $d < 1$, this probability is a.a.s. 0. **Example:** if $d = 0.99$, and $L = 1000$ this probability is about 0.00004. In other words, even if NeuPerm only displaces up to 1% of the parameters in each layer, an attack that uses even just 1000 parameters to hide information has almost no chance of succeeding. Of course, to mitigate such attacks, the defender can employ simpler mitigation techniques such as adding random noise to the parameters [3], [6].

E. Attacks With Error-Correction

We proceed to explore the more challenging case where the attack is error-resilient (e.g., MaleficNet [15]).

Assume the adversary's error-correcting code can correct $t' = \delta L$ errors with $0 < \delta < \frac{1}{2}$. Let $C = \sum_{k=1}^{L} X_k$ be the number of correctly placed payload bits; therefore we can say:

$$S > 0 \Leftarrow C \ge (1-\delta)L$$

We will use the additive Hoeffding bound [16] to get a probabilistic upper bound on S. Let

$$t \;:=\; (1-\delta)L - \mathbb{E}[C] \;\ge\; \big[(1-\delta) - d\big]L$$

The transition holds since that by Lemma V.1, $\mathbb{E}[C] \le dL$. Applying the additive Hoeffding bound to the independent $\{X_k\}$ gives, for $d < 1 - \delta$,

$$\begin{aligned}\Pr[C - \mathbb{E}[C] \ge t] &= \Pr[C \ge (1-\delta)L] \\ &\le \exp\Big(-\frac{2t^2}{L}\Big) \\ &\le \exp\big(-2\,[(1-\delta) - d]^2\, L\big).\end{aligned}$$

Thus a computationally unbounded adversary's success probability satisfies

$$\boxed{S \;\le\; d^L \;+\; \exp\big(-2\,[(1-\delta) - d]^2\, L\big), \quad d < 1 - \delta,}$$

where d^L bounds a blind guess of the permutation tuple P.

F. Limitation: Non-Permutable Architectures

An adversary may simply choose a network architecture in which $\mathcal{L}_{\mathrm{perm}} = \emptyset$ (e.g. a purely sequential model with tied weights, or certain RNNs lacking permutation symmetry). In that case NeuPerm cannot apply any randomization and $S = 1$. Thus practical deployments must ensure models contain at least one large permutable layer to obtain any security benefit.

VI. Neural Network Steganography Disruption Methods

For comparison, we implement other methods cited by previous works used to disrupt neural network steganography. All the different methods do so by slightly changing the parameters in a way that minimizes model performance degradation. Song et al., and Amit et al. [34], [3] suggest adding random normal noise to the model parameters - a simple and effective method. Additional options are fine-tuning [3], [40], [24], parameter pruning [15], and quantization [15], [6]. See Table II for a comparison of the methods. We consider the following questions: is the method quick, generic, does it not require retraining, does it not degrade performance, and whether or not it is effective on steganography attacks, StegoNet (SN), EvilModel (EM), and MaleficNet (MN). We mark answers to these questions like so: ○ = no, ◓ = partially, and ● = yes. Adding random noise is the simplest method; it only requires adding two matrices, and hence, it is very quick and the most generic, since it does not depend on different variables like model architecture, parameter type, etc. However, adding too much noise can degrade performance as empirical results show (see Section VIII-A). Fine-tuning is not quick, since it requires training deep learning models, which might take hours, and it is not generic, since it requires careful manual actions, tailored to the specific model architecture, task, etc. Parameter pruning and quantization both require fine-tuning to recover lost performance, so they include the downsides of fine-tuning. MaleficNet [15] establishes that adding random noise, parameter pruning, and fine-tuning are

not effective methods against it, but they note quantization **might** work without empirically proving that. NeuPerm is quick, as it only requires reordering the axes of the model parameter matrices in place. It does not require fine-tuning, and largely does not degrade model performance thanks to the permutation symmetry property. Finally, NeuPerm is partially generic, since it requires proving the permutation symmetry for different neural network types, and then implementing it as a program; however, there are a relatively small number of neural network architectures (MLP, CNN, RNN, ATTN, etc.), and we cover MLPs, CNNs, and ATTN in this work.

TABLE II
COMPARISON OF NEURAL NETWORK STEGANOGRAPHY DISRUPTION METHODS. WE REPORT THE EFFECTIVENESS OF THE METHODS ON PRIOR NEURAL NETWORK STEGANOGRAPHY WORKS: STEGONET (SN), EVILMODEL (EM), MALEFICNET (MN) [24], [40], [15]. EFFECTIVENESS MEANS WHETHER A METHOD CAN DISRUPT NEURAL NETWORK STEGANOGRAPHY TO SOME EXTENT, AND IS INDIVIDUALLY INDICATED BY THE ORIGINAL AUTHORS.

	Quick	Generic	Does not Require Retraining	Does not Degrade Performance	Effectiveness		
Method					SN [24]	EM [40]	MN [15]
Noise [3], [6]	●	●	●	◓	●	●	○
Pruning [15]	○	○	○	○	●	●	○
Fine-tuning [15]	○	○	○	○	●	●	○
Quantization [6], [15]	○	○	○	○	●	●	●
NeuPerm (Ours)	●	◓	●	●	●	●	●

VII. DATASETS

To evaluate NeuPerm, we aim to analyze two key properties: (1) Model performance after applying NeuPerm. (2) Survivability of state-of-the-art neural network steganography (MaleficNet) after applying NeuPerm.

Pre-trained CNN models: We used 3 well-known image classification CNN architectures: DenseNet, ResNet, and VGG. In particular, we use the DenseNet121, ResNet50, ResNet101, and VGG11 variants. For evaluating model performance, we use pre-trained PyTorch models trained on the ImageNet12 dataset, and we evaluate them on the validation split.

Pre-trained LLM models: We use the meta-llama/Llama-3.2-1B HuggingFace model with float16 parameters in our experiments. For evaluating model performance, we use the SQuAD benchmark.

Reproduced MaleficNet Stegomodels: We closely replicate MaleficNet [15] stegomodels by using the models, malware payloads, and datasets they used, along with their provided code. For the CNN models, we use 4 models: DenseNet121, ResNet50, ResNet101, and VGG11. For the CNN datasets, we use 2: ImageNet and CIFAR-10. For the malware, we use 9 samples downloaded from TheZoo [43]: Stuxnet, Destover, Asprox, Bladabindi, Zeus-Bank, EquationDrug, Kovter, Cerber, and Ardamax. Not all malware payloads fit in all models. This results in **46** replicated CNN MaleficNet stegomodels, and **9** replicated LLM MaleficNet stegomodels.

VIII. EXPERIMENTAL RESULTS

In this section, we present experimental results of NeuPerm and related baseline methods (see Section VI). We evaluate the methods in terms of 2 aspects: **(1) effect on model performance:** we consider methods with the least difference in model performance before and after applying them, the most optimal; we evaluate the models on validation tasks for this purpose. **(2) effectiveness against MaleficNet:** as we discussed in Section IV, all past neural network steganography works [24], [40] except MaleficNet [15] can be easily disrupted by introducing any change to the model parameters; therefore, we focus on evaluating the suggested method on MaleficNet, which stands as a non-trivial attack that does not have any defense work proven to work against it (the authors only claim without proof that model quantization will probably work). See Section VII for the data we used for our experiments (models, benchmarks, etc.). Experiment 1 (Section VIII-A) evaluates model performance, and Experiment 2 (Section VIII-B) evaluates effectiveness on NeuPerm.

A. Experiment 1: NeuPerm Does Not Degrade Performance

In this experiment, we analyze the effects of NeuPerm on model performance. Even though we proved NeuPerm does not change the neural network computation theoretically (see Section III), in practice, GPU computations are not fully deterministic, and floating numbers can not perfectly represent all real numbers; hence, we also assert that **NeuPerm does not degrade model performance** empirically. We analyze CNNs and an LLM model. See Table III. Our reproduced CNN models achieve similar accuracy on the ImageNet12 validation set. We can see NeuPerm (full) causes no difference in model accuracy for almost all the chosen CNN architectures. The VGG11 model got a 0.002% accuracy drop, which is negligible. Compared with NeuPerm, adding random normal noise to the parameters with a standard deviation of 0.0001 does degrade performance slightly, and as the standard deviation increases, the model performance is completely eroded. Pruning even 1% of the model parameters without retraining to regain accuracy has a dramatic negative effect. We can conclude that NeuPerm does not affect CNN model performance as the theory suggests. Moving on to the Llama-3.2-1B float16 model. See Table IV. Here we can observe that NeuPerm caused a 0.07±0.2 score difference compared to the baseline. As we anticipated, the theory guarantees that the neural network computes the same function, but it relies on various factors like the associativity of addition, etc., which are not modeled perfectly in computing, especially with GPU multiprocessing, and the fact that computer float values can not represent all mathematically possible real numbers. Nevertheless, we argue this is a negligible change in performance, and adding random noise or parameter pruning produced results with substantial variability, limiting the reliability of these methods compared with NeuPerm, which had a small standard deviation.

B. Experiment 2: NeuPerm Successfully Disrupts MaleficNet

In this experiment, we analyze how effectively NeuPerm disrupts CNN and LLM MaleficNet stegomodels, and we compare it to adding random noise, another disruption technique (see Section VI). According to the MaleficNet paper [14], given a MaleficNet stegomodel, it is proven that if the signal-to-noise ratio (SNR) is lower than 1, then the payload can not

TABLE III

MEAN VALIDATION ACCURACY (%) OF SELECT PYTORCH PRE-TRAINED CNN MODELS ON THE IMAGENET12 DATASET. WE REPORT THE ACCURACY CITED ON PYTORCH'S SITE, THE ACCURACY OF THE MODELS EVALUATED ON OUR MACHINE, AND THEN THE ACCURACY OF THOSE SAME MODELS AFTER APPLYING NEURAL NETWORK STEGANOGRAPHY DISRUPTION TECHNIQUES, INCLUDING NEUPERM (OURS), ADDING RANDOM NORMAL NOISE, AND PARAMETER PRUNING. RUNS ARE REPEATED 10 TIMES; THE STANDARD DEVIATION WAS LESS THAN 1E-1 IN ALL CASES, SO WE DO NOT REPORT IT. **ONLY FULL NEUPERM AND ADDING RANDOM NOISE WITH A 0.0001 STANDARD DEVIATION HAVE ACCEPTABLE PERFORMANCE DROPS.**

	PyTorch Reported Accuracy	Our Reproduced Accuracy	NeuPerm	Random Normal Noise Standard Deviation				Parameter Pruning Pruning Ratio	
				0.0001	0.001	0.01	0.1	1%	5%
DenseNet121	**74.434**	**74.434**	**74.434**	**74.38**	74.02	14.85	0.1	69.156	36.116
ResNet50	**80.858**	**80.844**	**80.844**	**80.816**	48.778	0.1	0.086	55.412	0.824
ResNet101	**81.886**	**81.904**	**81.904**	**81.9**	81.6	0.1	0.1	77.094	4.494
VGG11	**69.02**	**69.046**	**69.044**	**69.046**	68.95	61.35	0.1	59.826	45.474

TABLE IV

MEAN F1-SCORE (%) ± STANDARD DEVIATION OF LLAMA-3.2-1B LLM WITH FLOAT16 PARAMETERS ON THE SQUAD BENCHMARK. WE REPORT THE ACCURACY OF THE BASE MODEL EVALUATED ON OUR MACHINE, AND THEN THE ACCURACY OF THE SAME MODEL AFTER APPLYING NEURAL NETWORK STEGANOGRAPHY DISRUPTION TECHNIQUES, INCLUDING NEUPERM (OURS), ADDING RANDOM NORMAL NOISE, AND PARAMETER PRUNING. RUNS ARE REPEATED 10 TIMES. **ONLY FULL NEUPERM AND ADDING RANDOM NOISE WITH 0.0001 STANDARD DEVIATION HAVE ACCEPTABLE PERFORMANCE FLUCTUATIONS; NEUPERM HAD THE SCORES CLOSEST TO THE BASELINE.**

Base	NeuPerm	Random Normal Noise Standard Deviation				Prune Pruning Ratio	
		0.0001	0.001	0.01	0.1	1%	5%
18.09	**18.01 ± 0.2**	**17.19 ± 0.9**	22.0 ± 5.2	0	0	20.05 ± 6.1	12.68 ± 3.3

TABLE V

SNR OF IMAGENET12 CNN AND LLM MALEFICNET MODELS BEFORE AND AFTER APPLYING DISRUPTION TECHNIQUES. BASELINE (B) IS THE SNR OF THE MODEL AFTER ATTACKING IT WITH MALEFICNET. RANDOM NOISE (RN) IS THE SNR AFTER ADDING RANDOM NORMAL NOISE (BASELINE METHOD [3]) WITH A STANDARD DEVIATION OF 0.0001. FINALLY, NEUPERM (NP) (OUR SUGGESTED METHOD) IS THE SNR AFTER APPLYING FULL NEUPERM. **LOWER SNR IS BETTER, SNR LESS THAN 1 MEANS THE ATTACK IS DISRUPTED**. A "-" SIGN MEANS THE PAYLOAD IS TOO LARGE FOR THE GIVEN ARCHITECTURE. **TAKEAWAY:** NEUPERM SUCCESSFULLY DISRUPTS MALEFICNET, AND ADDING RANDOM NOISE DOES NOT.

	DenseNet121			ResNet50			ResNet101			VGG11			Llama-3.2-1B		
Malware	B	RN	NP	B	RN	NP	B	RN	NP	B	RN	NP	B	RN	NP
Stuxnet	5.1	5.1	**-8.7**	1.9	1.9	**-20.9**	5.4	5.4	**-15.6**	7.7	7.7	**-42.6**	7.5	7.5	**-1.7**
Destover	4.2	4.2	**-10.8**	1.9	1.9	**-25.0**	5.1	5.1	**-18.0**	8.1	8.1	**-24.5**	7.7	7.7	**-1.6**
Asprox	-	-	-	1.3	1.3	**-27.9**	5.0	5.0	**-16.5**	7.2	7.2	**-56.1**	6.9	6.9	**-1.4**
Bladabindi	-	-	-	2.2	2.2	**-24.8**	4.9	4.9	**-15.3**	7.7	7.7	**-19.5**	7.8	7.8	**-1.2**
Zeus-Bank	-	-	-	-	-	-	4.8	4.8	**-17.3**	7.4	7.4	**-30.7**	7.5	7.5	**-0.9**
Eq.Drug	-	-	-	-	-	-	3.9	3.9	**-14.6**	7.1	7.1	**-44.2**	6.9	6.9	**-1.4**
Kovter	-	-	-	-	-	-	4.2	4.2	**-21.8**	7.1	7.1	**-20.4**	7.8	7.8	**-1.2**
Cerber	-	-	-	-	-	-	4.4	4.4	**-20.9**	6.9	6.9	**-20.7**	7.0	7.0	**-2.1**
Ardamax	-	-	-	-	-	-	-	-	-	5.9	5.9	**-27.5**	7.2	7.2	**-0.9**

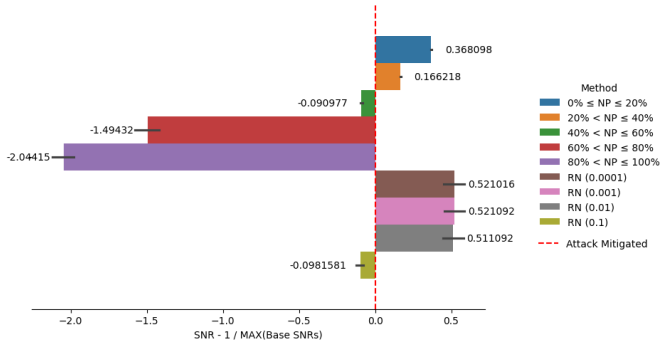

Fig. 3. Mean normalized SNR difference of all CNN MaleficNet stegomodels after applying disruption techniques. We use partial NeuPerm (NP) methods: $a\% \leq NP \leq b\%$ denotes NP methods that changed between $a\%$ and $b\%$ of the model parameters. Random Noise (RN) (f) denotes adding random normal noise with standard deviation f. **Values less than 0 mean the attack was mitigated**. **Takeaway:** NeuPerm was effective if it changed at least 40% of the parameters, and adding Random Noise is only effective with a standard deviation of 0.1, which destroyed model performance (see Section VIII-A); hence, **NeuPerm is the only effective and practical method to disrupt MaleficNet.**

be successfully extracted. Therefore, we use SNR to measure how disruptive applied techniques are, and we regard SNR values of less than 1 to mean the attack has been successfully mitigated. See Table V, we focus on the CNN ImageNet12 and LLM MaleficNet stegomodels (see Section VII). We can see that full NeuPerm caused the SNR to become negative, which strongly asserts that the attack was disrupted and the payload can not be extracted. Moreover, adding random noise with a standard deviation of 0.0001 had little to no effect on the SNR. This aligns with the empirical results shown in MaleficNet [15], [14], which additionally asserts that fine-tuning and parameter pruning are also ineffective. We also provide a more in-depth comparison of NeuPerm (NP) with varying degrees of percent of parameters changed, and adding random noise with increasing standard deviation. See Fig. 3. We see that applying NeuPerm schemes that only permute up to 40% of parameters is insufficient for fully mitigating MaleficNet, and adding random noise with at least 0.1 standard deviation mitigates MaleficNet. We recall that adding random noise with a standard deviation of more than 0.0001 significantly hinders model performance. Hence, we can conclude that adding random noise is not a reasonable way to disrupt MaleficNet, and NeuPerm is.

IX. LIMITATIONS AND FUTURE WORK

Our suggested NeuPerm method is based on the theoretical property of permutation symmetry. However, not every neural network might exhibit this, for example, a CONV-CONV block with a residual connection in the middle, i.e., $f(x) = CONV_2(CONV_1(x) + x)$. This means, in essence, NeuPerm is not applicable in these scenarios. But, one possible workaround might be the introduction of a $PERMUTE$ neural network layer, which permutes the axes of its input; these can be manually inserted into the neural network architecture. They might add a performance overhead to the overall computation. Nevertheless, we demonstrated that NeuPerm can be applied to CNNs that have residual connections like DenseNet. Since NeuPerm is dependent on the permutation symmetry property, to extend it to different neural network architectures, one has to assert that the property holds, by proving it or otherwise. Moreover, this also means NeuPerm is only applicable to neural networks that have the permutation symmetry property. In this work, we focus on permutation symmetry, however, there are more types of symmetry that model parameters can exhibit [44]. Future works can extend the methodology to also take advantage of them. Finally, in theory, to avoid dealing with NeuPerm, attackers can embed

their payloads in neural network architectures that have fewer parameter permutation symmetries [23]; however, these kinds of models are not popular, and we successfully used NeuPerm in the widely-used CNN models and the Llama-3.2-1B, which is very popular (2.1M downloads last month on HuggingFace) at the time of writing this paper.

X. Conclusion

In this paper, we introduce NeuPerm, a simple yet effective method for disrupting neural network steganography. Our method is based on a theoretical property that ensures model performance does not change after applying it. This is very significant, as all related methods do affect model performance, and sometimes very drastically. Moreover, our method is empirically shown to disrupt resilient error-correcting steganography such as MaleficNet, which is only achieved otherwise (presumably) using quantization, a complex method that requires retraining, expertise, and resources, and which is highly specific to model architecture and hardware. Our work is the first to develop a method that explicitly disrupts MaleficNet; past work by Dubin [6] only focused on basic LSB substitution attacks like StegoNet and EvilModel [24], [40], which are very easily disrupted. Furthermore, we develop and evaluate our methods on MLPs, CNNs, and LLMs; past works only focused on CNNs.

References

[1] Samuel K. Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. Git re-basin: Merging models modulo permutation symmetries, 2023.

[2] Ahmed A AlSabhany, Ahmed Hussain Ali, Farida Ridzuan, AH Azni, and Mohd Rosmadi Mokhtar. Digital audio steganography: Systematic review, classification, and analysis of the current state of the art. *Computer Science Review*, 38:100316, 2020.

[3] Guy Amit, Mosh Levy, and Yisroel Mirsky. Transpose attack: Stealing datasets with bidirectional training, 2023.

[4] Junyi Chai, Hao Zeng, Anming Li, and Eric WT Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[5] Li Deng and Yang Liu. *Deep learning in natural language processing*. Springer, 2018.

[6] Ran Dubin. Disarming attacks inside neural network models. *IEEE Access*, 11:124295–124303, 2023.

[7] Rahim Entezari, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. The role of permutation invariance in linear mode connectivity of neural networks, 2022.

[8] Andre Esteva, Katherine Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. Deep learning-enabled medical computer vision. *NPJ digital medicine*, 4(1):5, 2021.

[9] Joseph Gallian. *Contemporary abstract algebra*. Chapman and Hall/CRC, 2021.

[10] Daniel Gilkarov and Ran Dubin. Model x-ray: Detection of hidden malware in ai model weights using few shot learning, 2024.

[11] Daniel Gilkarov and Ran Dubin. Steganalysis of ai models lsb attacks. *IEEE Transactions on Information Forensics and Security*, 19:4767–4779, 2024.

[12] Palash Goyal, Sumit Pandey, and Karan Jain. Deep learning for natural language processing. *New York: Apress*, 2018.

[13] Robert Hecht-Nielsen. On the algebraic structure of feedforward network weight spaces. In Rolf ECKMILLER, editor, *Advanced Neural Computers*, pages 129–135. North-Holland, Amsterdam, 1990.

[14] Dorjan Hitaj, Giulio Pagnotta, Fabio De Gaspari, Sediola Ruko, Briland Hitaj, Luigi V. Mancini, and Fernando Perez-Cruz. Do you trust your model? emerging malware threats in the deep learning ecosystem, 2024.

[15] Dorjan Hitaj, Giulio Pagnotta, Briland Hitaj, Luigi V. Mancini, and Fernando Perez-Cruz. Maleficnet: Hiding malware into deep neural networks using spread-spectrum channel coding. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 425–444, Cham, 2022. Springer Nature Switzerland.

[16] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[17] HuggingFace. HuggingFace Hub, 2022. Accessed: 2022-12-19.

[18] Mohammadreza Iman, Hamid Reza Arabnia, and Khaled Rasheed. A review of deep transfer learning and recent advancements. *Technologies*, 11(2):40, 2023.

[19] Andreas Kamilaris and Francesc X Prenafeta-Boldú. Deep learning in agriculture: A survey. *Computers and electronics in agriculture*, 147:70–90, 2018.

[20] Konstantinos Karampidis, Ergina Kavallieratou, and Giorgos Papadourakis. A review of image steganalysis techniques for digital forensics. *Journal of information security and applications*, 40:217–235, 2018.

[21] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018.

[22] Derek Lim, Haggai Maron, Marc T. Law, Jonathan Lorraine, and James Lucas. Graph metanetworks for processing diverse neural architectures, 2023.

[23] Derek Lim, Theo Putterman, Robin Walters, Haggai Maron, and Stefanie Jegelka. The empirical impact of neural parameter symmetries, or lack thereof. *Advances in Neural Information Processing Systems*, 37:28322–28358, 2024.

[24] Tao Liu, Zihao Liu, Qi Liu, Wujie Wen, Wenyao Xu, and Ming Li. Stegonet: Turn deep neural network into a stegomalware. In *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, page 928–938, New York, NY, USA, 2020. Association for Computing Machinery.

[25] Mohammed Abdul Majeed, Rossilawati Sulaiman, Zarina Shukur, and Mohammad Kamrul Hasan. A review on text steganography techniques. *Mathematics*, 9(21):2829, 2021.

[26] Adam C Mater and Michelle L Coote. Deep learning in chemistry. *Journal of chemical information and modeling*, 59(6):2545–2559, 2019.

[27] Seonwoo Min, Byunghan Lee, and Sungroh Yoon. Deep learning in bioinformatics. *Briefings in bioinformatics*, 18(5):851–869, 2017.

[28] MITRE. Mitre ATLAS, 2025.

[29] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[30] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

[31] OWASP. OWASP Top 10 for LLM Applications 2025, 2025.

[32] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[33] Pramila P Shinde and Seema Shah. A review of machine learning and deep learning applications. In *2018 Fourth international conference on computing communication control and automation (ICCUBEA)*, pages 1–6. IEEE, 2018.

[34] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and Communications Security*, pages 587–601, 2017.

[35] Nandhini Subramanian, Omar Elharrouss, Somaya Al-Maadeed, and Ahmed Bouridane. Image steganography: A review of the recent advances. *IEEE access*, 9:23409–23423, 2021.

[36] TensorFlow. TensorFlow Hub, 2022. Accessed: 2022-12-19.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[39] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018(1):7068349, 2018.
[40] Zhi Wang, Chaoge Liu, and Xiang Cui. Evilmodel: Hiding malware inside of neural network models. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7, 2021.
[41] Zhi Wang, Chaoge Liu, Xiang Cui, Jie Yin, and Xutong Wang. Evilmodel 2.0: Bringing neural network models into malware attacks. *Computers & Security*, 120:102807, 2022.
[42] Stephen Wu, Kirk Roberts, Surabhi Datta, Jingcheng Du, Zongcheng Ji, Yuqi Si, Sarvesh Soni, Qiong Wang, Qiang Wei, Yang Xiang, et al. Deep learning in clinical natural language processing: a methodical review. *Journal of the American Medical Informatics Association*, 27(3):457–470, 2020.
[43] ytisf. theZoo - A Live Malware Repository, 2025. Accessed: 2025-04-28.
[44] Binchi Zhang, Zaiyi Zheng, Zhengzhang Chen, and Jundong Li. Beyond the permutation symmetry of transformers: The role of rotation for model fusion, 2025.
[45] Na Zhao, Kejiang Chen, Chuan Qin, Yi Yin, Weiming Zhang, and Nenghai Yu. Calibration-based steganalysis for neural network steganography. In *Proceedings of the 2023 ACM Workshop on Information Hiding and Multimedia Security*, pages 91–96, 2023.
[46] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

APPENDIX

In this section, we provide mathematical proofs of neural network parameter permutation symmetry for common neural network "blocks".

To prove permutation symmetry, it is sufficient to prove that swapping two units keeps the computation identical. This is true because a permutation can be expressed as a series of pairwise swaps [9].

A. Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP), often referred to as a feedforward neural network or fully connected network, is a foundational neural network architecture. It is composed of an input layer, one or more hidden layers of computational units, and an output layer [32].

1) Formal Computation of an MLP: Let $f(\cdot)$ be an MLP with L hidden layers. Let $x \in \mathbb{R}^{d_0}$ be the input vector, and let d_ℓ be the number of units in layer ℓ (layer 0 is the input layer).

For each layer $\ell = 1, 2, \ldots, L$, define:

- $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ – the weight matrix,
- $b^{(\ell)} \in \mathbb{R}^{d_\ell}$ – the bias vector,
- $\sigma^{(\ell)}(\cdot)$ – the (elementwise) activation function,
- $z^{(\ell)}$ – the pre-activation at layer ℓ,
- $h^{(\ell)}$ – the post-activation (output) at layer ℓ.

We set $h^{(0)} = x$ and then, for each layer ℓ, the feedforward computations are:

$$z^{(\ell)} = W^{(\ell)}\, h^{(\ell-1)} + b^{(\ell)},$$

$$h^{(\ell)} = \sigma^{(\ell)}\big(z^{(\ell)}\big).$$

Finally, the network's output $f(x)$ is given by:

$$f(x) = h^{(L)}$$

2) MLP Swap: We define an MLP swap as a swap between two units inside a hidden layer, and we argue that this swap does not change the MLP computation at all. In terms of the parameter weights/bias matrices/vectors, a swap between the i'th and j'th units in some layer l means swapping the i'th and j'th rows in $W^{(l)}$, and $b^{(l)}$; and also swapping the i'th and j'th columns in $W^{(l+1)}$. Formally, we express a matrix W with rows i, j swapped like so:

$$swaprows(W, i, j) = \begin{cases} W_j, & \text{if } k = i, \\ W_i, & \text{if } k = j, \\ W_k, & \text{otherwise.} \end{cases}$$

Where W_i denotes the i'th row of W. Similarly, we can define $swapcols(W, i, j)$.

Claim 1.1: Let $A \in \mathbb{R}^{n \times m}$, $x \in \mathbb{R}^{m \times 1}$, $1 \le i < j \le m$, then:

$$swapcols(A, i, j)swaprows(x, i, j) = Ax$$

Claim 1.1 Proof:

$$swapcols(A, i, j)swaprows(x, i, j) =$$

$$= \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1j}x_j + \cdots + a_{1i}x_i + \cdots + a_{1m}x_m \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nj}x_j + \cdots + a_{ni}x_i + \cdots + a_{nm}x_m \end{bmatrix}$$

$$= \begin{bmatrix} \sum_{k=1}^{m} a_{1k}x_k \\ \vdots \\ \sum_{k=1}^{m} a_{nk}x_k \end{bmatrix} = Ax$$

We proceed to prove the MLP swap symmetry property.

Claim 1.2 (MLP swap symmetry property): let l_1 and $l_2 := l_1 + 1$ be two consecutive hidden layers in some MLP $f(\cdot)$. Layer l_1 has $d_{l_1} := d$ units: let $1 \le i < j \le d$ be indexes of two units in layer l_1. We set:

$$\begin{aligned} W_1 &:= W^{(l_1)} \\ b &:= b^{(l_1)} \\ W_2 &:= W^{(l_2)} \\ W_1' &:= swaprows(W_1, i, j) \\ b' &:= swaprows(b, i, j) \\ W_2' &:= swapcols(W_2, i, j) \end{aligned}$$

The pre-activations and post-activations before and after the swap are:

$$\begin{aligned} z^{(l_1)} &= W_1\, h^{(l_1-1)} + b \\ h^{(l_1)} &= \sigma^{(l_1)}(z^{(l_1)}) \\ z^{(l_2)} &= W_2\, h^{(l_1)} + b \\ z'^{(l_1)} &= W'_1\, h^{(l_1-1)} + b' \\ h'^{(l_1)} &= \sigma^{(l_1)}(z'^{(l_1)}) \\ z'^{(l_2)} &= W'_2\, h'^{(l_1)} + b^{(l_2)} \end{aligned}$$

then $z^{(l_2)} = z'^{(l_2)}$ holds.

Claim 1.2 Proof: We wish to show that $z^{(l_2)} = z'^{(l_2)}$, i.e. we want:

$$\begin{aligned} z^{(l_2)} = z'^{(l_2)} &\Rightarrow W_2\, h^{(l_1)} + b^{(l_2)} = W'_2\, h'^{(l_1)} + b^{(l_2)} \\ &\Rightarrow W_2\, h^{(l_1)} = W'_2\, h'^{(l_1)} \end{aligned}$$

Let us show that $z'^{(l_1)} = swaprows(z^{(l_1)}, i, j)$. Let r_i be the ith row of W_1, and r'_i be the ith row of W'_1.

$$z^{(l_1)} = \begin{bmatrix} r_1 h^{(l_1-1)} + b_1 \\ r_2 h^{(l_1-1)} + b_2 \\ \vdots \\ r_i h^{(l_1-1)} + b_i \\ \vdots \\ r_j h^{(l_1-1)} + b_j \\ \vdots \\ r_d h^{(l_1-1)} + b_d \end{bmatrix}$$

$$z'^{(l_1)} = \begin{bmatrix} r'_1 h^{(l_1-1)} + b'_1 \\ r'_2 h^{(l_1-1)} + b'_2 \\ \vdots \\ r'_i h^{(l_1-1)} + b'_i \\ \vdots \\ r'_j h^{(l_1-1)} + b'_j \\ \vdots \\ r'_d h^{(l_1-1)} + b'_d \end{bmatrix} = \begin{bmatrix} r_1 h^{(l_1-1)} + b_1 \\ r_2 h^{(l_1-1)} + b_2 \\ \vdots \\ r_j h^{(l_1-1)} + b_j \\ \vdots \\ r_i h^{(l_1-1)} + b'_i \\ \vdots \\ r_d h^{(l_1-1)} + b_d \end{bmatrix}$$

$$= swaprows(z^{(l_1)}, i, j)$$

Since $z'^{(l_1)} = swaprows(z^{(l_1)}, i, j)$, it means that $h'^{(l_1)} = swaprows(h^{(l_1)}, i, j)$ holds:

$$\begin{aligned} h'^{(l_1)} = \sigma^{(l_1)}(z'^{(l_1)}) = \\ = \sigma^{(l_1)}(swaprows(z^{(l_1)}, i, j)) \\ = swaprows(\sigma^{(l_1)}(z^{(l_1)}), i, j) \\ = swaprows(h^{(l_1)}, i, j) \end{aligned}$$

Finally, we get:

$$W'_2 h'^{(l_1)} = swapcols(W_2, i, j) swaprows(h^{(l_1)}, i, j) = W_2 h^{(l_1)}$$

Since a swap keeps the MLP computation the same, we get that permuting the units also does so.

B. Multi-Head Self-Attention (MHSA)

The multi-head self-attention block is the foundational component of transformer [38] neural network architectures, and in particular, LLMs.

1) Formal Computation of Multi-Head Attention: The original work defines a multi-head attention block generally, with independent Q, K, V input matrices. Here, we will focus on self-attention ($Q = K = V$), which is what LLMs use. We start with a single attention block: Let $h \in \mathbb{N}$ be the number of attention heads, $d_{model} \in \mathbb{N}$, the model embedding dimension. Set $d := d_{model}/h$.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V$$

Multi-head self-attention is defined like so:

$$MultiHead(X) = W^O Concat(head_1, ..., head_h)$$
Where $head_i = Attention(W_i^Q X, W_i^K X, W_i^V X)$

Where $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d \times d_{model}}, W^O \in \mathbb{R}^{d_{model} \times d_{model}}$ are the parameter matrices of the multi-head attention block. Set $swaprows(\cdot, i, j) := sr(\cdot), swapcols(\cdot, i, j) := sc(\cdot)$.

2) MHSA Swap: We argue that we can perform $sr(W_k^Q), sr(W_k^K), sr(W_k^V)$, $swapcols(W^O, kh + i, kh + j)$, for some k (i.e., we swap 2 units in a specific head), and the computation will stay the same. Let us now reframe the previously introduced sr, and sc functions as rotation matrices. Let i, j be some indexes we want to swap; define the rotation matrix $R_{i,j}$:

$$R_{i,j} = swaprows(I, i, j)$$

where I is the identity matrix. We have the following properties:

$$\begin{aligned} R_{i,j}A &= sr(A) \\ AR_{i,j} &= sc(A) \\ R_{i,j}^2 &= I \\ R_{i,j}^T &= R_{i,j} \\ f(R_{i,j}A) &= R_{i,j} f(A) \\ f(AR_{i,j}) &= f(A) R_{i,j} \end{aligned}$$

Where A is any matrix, and f is a monotone increasing function. Note: $softmax$, and $f(x) = \frac{1}{c}$ for any positive c are monotone increasing functions. Let us look at the attention computation after performing the swaps, set $A := W_k^Q, B := W_k^K, C := W_k^V, R := R_{i,j}$. Denote $head'_k$ and $MultiHead'$ as the versions of the functions after the swaps.

$$\begin{aligned} head'_k &:= Attention(RW_k^Q X, RW_k^K X, RW_k^V X) \\ (RAX)((R(BX))^T) &= RAX((BX)^T R^T) \\ &= RAX((BX)^T R) \\ &\Downarrow \\ head'_k &= Rsoftmax(\frac{AX(BX)^T}{\sqrt{d}})RRCX \\ &= Rhead_i = sr(head_k) \\ &\Downarrow \\ MultiHead'(X) &= \\ swapcols(W^O, kh + i, kh + j) &Concat(head_1, ..., sr(head_k), ..., head_h)) \\ &= Multihead(X) \end{aligned}$$