

Permutation-Based Stegomalware in Large Language Models: Threats and Countermeasures

Danny Wood
Fuzzy Labs

danny@fuzzylabs.ai

James Stringer
Fuzzy Labs

james@fuzzylabs.ai

Abstract

The difficulty of training large language models (LLMs), together with their ubiquity, raises the threat of stegomalware, where malicious payloads are embedded into model weights. Recent work has demonstrated the use of permutation symmetry in model weights to mitigate these threats, but failed to show neutralization of stegomalware across all weights for LLMs. In this paper, we demonstrate the full potential of behavior-preserving symmetries as a defense against stegomalware, as well as the risks these symmetries pose when exploited by attackers.

For stegomalware neutralization, we improve upon previous work, demonstrating that it is possible to select permutations which displace all model parameters. This contrasts with previous methods which left a significant percentage of weights unaltered in LLMs. When used in an attack, we show that permutation symmetries can encode malware into the weights of a model in a way that is theoretically lossless, requires no retraining after encoding, and needs no payload-specific information in the extraction script—a combination of characteristics not previously seen in any single method.

While theoretically lossless, permutation can in practice alter model behavior due to the accumulation of numerical error. We therefore quantify the loss in model performance associated with applying these methods, for both attack and defense, showing it to be minimal.

1 Introduction

Large language models are an increasingly vital part of many software products, but for most academic, personal, or even industrial uses, training a custom large language model from scratch is infeasible. This means that most developers and end users rely on pre-trained models from online repositories such as Hugging Face. This therefore creates a supply chain risk, as malicious actors can use these models as an attack vector.

Attacks on the supply chain for machine learning tools and software are becoming increasingly prevalent, both through conventional means such as compromising trusted packages [8] and more AI-specific means such as serving compromised models on Hugging Face. This kind of supply chain attack was listed as one of OWASP’s top ten risks for LLMs and GenAI Apps [21] and is likely to increase in prevalence as the demand for AI in products grows.

Real world attacks using compromised models thus far typically hide malicious payloads within code and data separate from the model weights themselves. However, recent research has demonstrated that it is possible for attackers to make their malicious code harder to detect through the use of steganography, creating malware which is hidden within the structures of the neural network itself [16, 25, 14]. Existing work has shown that neural networks make appealing targets for this kind of malware, since the high entropy and high fault tolerance of neural network weights allow for the insertion of significantly sized payloads with little drop in model performance. Steganographic malware has already proven both feasible and capable of evading

detection on other online software hubs, such as smartphone app stores [23], suggesting a need to understand and address these vulnerabilities in this new domain.

Recent work has begun to develop defenses against these attacks by exploiting functional invariances in neural network architectures, shuffling weights to disrupt payload extraction in specific constructions [24, 12]. This kind of functional invariance has also been used to watermark models for intellectual-property protection [11] and to analyze the structure of machine learning models [2].

As we will show, these invariances also pose a threat: they can be used not only to neutralize stegomalware but also to embed it. Embedding a payload in a large language model this way has significant benefits for the attacker: in exact arithmetic it preserves model behavior completely, requires no retraining, and needs no payload-specific extraction script. Existing attacks in the literature all sacrifice at least one of these three properties.

Fortunately, the attack and countermeasures in this scenario are complementary. While we are able to demonstrate that it is possible to store significant amounts of information in a large language model, we also show that the attack can be neutralized by randomly permuting the weight matrices of the network. Previous work relies on permutations whose effects are localized, confined within individual components or layers. We show that, as well as these localized symmetries, there are *global* permutations that propagate consistently across every layer. Composing the two, along with careful selection of the permutation to apply, shuffles 100% of the parameters in an LLM, which is sufficient for disruption of all known attacks in the literature.

Our contributions are as follows:

- We show how *global* permutation symmetries, propagating across the whole network, can be exploited to neutralize stegomalware in 100% of the parameters of a large language model, beating the previous state of the art which managed less than two thirds of all parameters.
- We introduce PermaNet, a novel, (theoretically) lossless method of embedding hidden information in neural networks.
- We demonstrate the efficacy of this neutralization, showing complete disruption of payload extraction on existing stegomalware encoding techniques, and complete removal of PermaNet-encoded payloads.
- We test our methods of malware embedding and neutralization, verifying a minimal reduction in model performance. This is less than the effects of quantization levels considered to be ‘near lossless’, and disappears as the level of numerical precision at inference is increased.

2 Threat Model

The threat model we define aligns closely with those described in [16] and [12]; this model is representative of how threats are likely to surface when embedded in pre-trained neural networks distributed to consumers through third party distribution platforms like Hugging Face. We define the following personas.

End user. The end user of this attack is any consumer of pre-trained neural networks which are distributed through third party platforms. We do not restrict the end user to a non-expert persona, as ML experts frequently consume pre-trained models for experimentation or deployment. The end user may either be an independent actor or part of a larger organization, and may deploy the model either on a local device or in a cloud environment. We assume that the end user stores and deploys the model as is, without performing additional transformations such as fine-tuning or quantization.

Adversary. We define the adversary as a third party provider of malicious pre-trained models. Unofficial

providers are common on distribution platforms and among them malicious providers are largely indistinguishable from benign ones. The adversary creates a malicious neural network and advertises it on a third party distribution platform. They may associate the malicious model with a trusted provider, such as through claiming their malicious model as a fine-tune or quantization of a trusted model, or by masquerading as a trusted organization, creating malicious spoofs of popular models. The goal of the adversary is to achieve remote code execution or a similar exploit on the end user’s machine using a payload embedded in their malicious model. They must fulfill several conditions in order to achieve this goal:

1. Maintain the performance of the malicious model relative to a benign model, both in order to evade detection of malicious behavior and to avoid reduced desirability of a degraded model.
2. Avoid detection with traditional anti-malware tools using methods like encoded string detection.
3. Decode and execute the malicious payload during a stage in the normal deployment of the model, without any additional access to the end user’s machine. Typically, this will occur during the deserialization of the model, such as through depickling.

We suppose that, in order to fulfill these conditions, the adversary uses the PermaNet method described in this work to store a malicious payload in the ordering of their model’s weights. PermaNet is the steganographic technique rather than a complete attack; we consider trigger mechanisms to be outside the scope of this work. As in prior work, we assume that a small *bootstrap* routine runs when the model is loaded. There have been several real-world vulnerabilities that could be used for this purpose, for instance through the well-known pickle deserialization vector, or in code packaged with the model weights. The bootstrap reads the model’s own weight tensors, runs the PermaNet decoder to recover the payload bytes, and executes them. PermaNet’s bootstrap consists of payload-agnostic tensor-manipulation operations, carrying no malicious bytes of its own. This attracts less suspicion than the payload it reconstructs, meaning that signature-based scanning will not detect the payload, only more benign-looking tensor operations. We discuss this in more detail in Section 8.

In Suarez-Tangil et al. [23], three types of stegomalware are described, depending upon whether the payload, extraction method and cryptographic key for decoding are present on the target system. The attacks and countermeasures which we describe in this paper are agnostic to which scenario is considered. PermaNet is not intended as a full end-to-end attack; instead, it is a specific steganographic method that an attacker might employ on neural network architectures, following in the spirit of other neural network steganography research like MaleficNet [14].

3 Permutation-Based Attacks

In this section, we show how it is possible to embed a potentially malicious payload into a large language model by exploiting permutation symmetries in the model parameters. Our argument has the following structure:

1. We show that information can be encoded by reordering the rows or columns of a given matrix.
2. We demonstrate how invariances in the structure of an LLM allow for permutation of parameter matrices while preserving model functionality.
3. We show how the two points above can be combined to store a payload in the weights of an LLM while perfectly preserving model functionality—assuming perfect precision arithmetic—and without requiring that we know the original order of the rows/columns.
4. Finally, we calculate how much data this allows to be encoded in real-world models, showing that it is substantial enough to encode common malware variants.

3.1 Encodable Bits

For a matrix with n unique rows, there are $n!$ possible orderings. Given complete freedom to permute those rows as we like, it is therefore possible to encode $\lfloor \log_2 n! \rfloor$ bits of information into that ordering. Intuitively, for a matrix with $n \leq 26$ rows, we label the rows with a canonical ordering as $A, B, C, \dots$. We may then list all possible permutations in alphabetical order, assigning each permutation a number between 0 and $n! - 1$ based on its position in that list. This allows us to communicate that number to a recipient who knew the matrix's original order by sending them the matrix whose rows are arranged according to the correspondingly ranked permutation. Equivalently, this can be thought of as encoding bits in the binary representation of the number, including leading zeros. This process is shown in Figure 1.

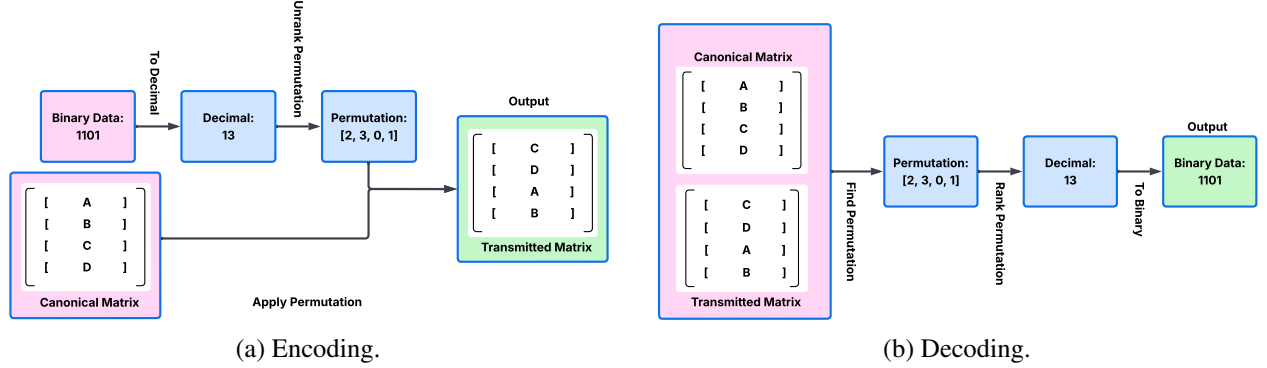


Figure 1: Visualization of the method for encoding and decoding information in the ordering of the rows of a matrix. Left: Binary data is converted into a numerical value, then encoded as a permutation. That permutation is then applied to the rows of the matrix before transmission. Right: The recipient calculates the permutation required to transform the canonical matrix into the received one. They then calculate the permutation's rank and use this to reconstruct the binary data.

In practice, the recipient does not need to know the original matrix beforehand; they only need to have previously agreed upon a procedure to choose a canonical ordering of rows of a matrix, that is, a method for constructing a bijection between the matrix's rows and the set $\{0, 1, \dots, n-1\}$. The sender and recipient also need efficient ways of calculating the rank of a given permutation in the list of all possible $n!$ permutations, as well as the ability to carry out the inverse of that operation.

For the canonical ordering, we deterministically order the rows: for instance by hashing them, or by sorting each row's entries and comparing rows at their first differing entry. We prefer the latter, as it allows us to establish a canonical ordering of the rows which is invariant to permutations of the columns, allowing us to encode information in both orderings at once.

Although the rank and its inverse can be computed using the lexicographical ordering, we instead use a different ordering that is simpler to rank and unrank in linear time [20]. Further to this, we modify this algorithm to derive a non-recursive variant, allowing us to handle large matrices without reaching Python's recursion-depth limit, as detailed in Appendix C.

Calculating the information storage capacity of the permutation of n objects requires computing $\log_2 n!$. In our preliminary investigation, we encountered issues converting $n!$ to a float for this operation, due to its size. Instead, we make use of the approximation

$$\log_2 n! \approx \frac{\frac{1}{2} \ln 2\pi n + n \ln n - n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2},$$

which is a variation on a well-known bound for $n!$. Furthermore, we can verify that this approximation costs at most one bit. The proofs of all results in this section are deferred to Appendix A.

Proposition 1. *For $n \geq 1$, we have*

$$\lfloor \log_2 n! \rfloor - 1 \leq \left\lfloor \frac{\frac{1}{2} \ln 2\pi n + n \ln n - n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} \right\rfloor \leq \lfloor \log_2 n! \rfloor.$$

Hence, using this approximation in place of the true value costs at most a single bit of storage. We conjecture that for $n > 2$ the lower bound could be tightened and that $\log_2 n!$ always has the same integer part as our approximation. Through elementary computation, it can be seen that this grows at a rate of $\mathcal{O}(n \log n)$, so the number of bits that can be stored grows super-linearly with the number of rows of a matrix that can be permuted.

3.2 Exploiting Invariances in a Transformer

We will now show that there is an invariance in standard transformer architectures which allows for permutations to be applied to all parameter matrices in the model while preserving the model’s function.

Before showing the full case of transformer models, we consider how such an invariance works in a single-layer MLP. Recall that for an $m \times n$ matrix W , permutation of rows can be achieved by construction of a permutation matrix P , with the property that $P^{-1} = P^\top$.

For input $x \in \mathbb{R}^d$:

$$\text{MLP}(x; W_1, W_2) = W_2 \text{ReLU}(W_1 x), \tag{1}$$

where $W_1 \in \mathbb{R}^{m \times d}$, $W_2 \in \mathbb{R}^{d \times m}$, and $\text{ReLU}(z) = \max(0, z)$ applied elementwise. Let $P \in \mathbb{R}^{m \times m}$ be a permutation matrix and define $W'_1 = PW_1$ and $W'_2 = W_2 P^\top$. Using the fact that element-wise functions commute with permutations of elements in a vector, we get

$$\begin{aligned} \text{MLP}(x; W'_1, W'_2) &= W'_2 \text{ReLU}(W'_1 x) = W_2 P^\top \text{ReLU}(PW_1 x) \\ &= W_2 P^\top P \text{ReLU}(W_1 x) = W_2 \text{ReLU}(W_1 x) = \text{MLP}(x; W_1, W_2). \end{aligned}$$

That is, applying the permutation P and its transpose inside the network in the right way preserves the network’s behavior. This gives us a family of $m!$ MLPs which are functionally equivalent but have different parameters. Like most MLP layers in modern LLMs, our example does not include bias terms, but the same principle would apply to a network including biases if we apply the same permutation to biases of the first layer.

The use of this simple example has been demonstrated in [12, 24] to neutralize malware hidden in the weights of MLPs. We will show how a more comprehensive permutation, first observed in Fernandez et al. [11], can be used to permute all the weights in an LLM while preserving model functionality. We will also show extension of this method to modern mixture-of-experts (MoE) architectures.

To start, we show a set of permutations of parameters for each self-attention and MLP layer of an LLM architecture that results in equivariance: i.e., permutation of the inputs to the layer results in an output that is equivalent to the original output up to the same permutation. We will then demonstrate that applying that same permutation to the input and output embedding weights results in invariance of the overall architecture.

We consider a multi-head self-attention on a sequence of length n with embedding dimension d . Let $X \in \mathbb{R}^{n \times d}$ be the embeddings of the input sequence, an attention head performs the function:

$$\text{Att}(X; \theta_{\text{att}}) = \text{Softmax}\left(\frac{XW_QW_K^\top X^\top}{\sqrt{d_k}}\right) XW_V, \quad (2)$$

where $W_Q \in \mathbb{R}^{d \times d_k}$, $W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$ are the respective query, key, and value weight matrices, for some query and value dimension values d_k and d_v . We write $\theta_{\text{att}} = \{W_Q, W_K, W_V, W_O\}$ for the full set of attention weight matrices, where $W_O \in \mathbb{R}^{hd_v \times d}$ is the output projection matrix. The values of the outputs of multiple heads are concatenated and projected back down to the embedding dimension:

$$\begin{aligned} \text{MHA}(X; \theta_{\text{att}}) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) W_O \\ \text{head}_i &= \text{Att}\left(X; \theta_{\text{att}}^{(i)}\right) \end{aligned} \quad (3)$$

Let $P \in \mathbb{R}^{d \times d}$ be a permutation matrix and define $W'_Q := P^\top W_Q$, $W'_K := P^\top W_K$, $W'_V := P^\top W_V$ and $W'_O := W_O P$. We write $\theta'_{\text{att}} = \{W'_Q, W'_K, W'_V, W'_O\}$

Proposition 2. *Multi-head attention is equivariant under permutation of input features, i.e., $\text{MHA}(XP; \theta'_{\text{att}}) = \text{MHA}(X; \theta_{\text{att}})P$.*

In our definition of multi-head attention, we omit Rotary Position Embedding (RoPE) [22], which applies rotations to the query and key activations to encode sequence position. This choice is for simplicity of exposition, and does not affect the proof since the rotation is on the internal dimensions of the query and key vectors, not on the permuted embedding dimension.

Next we consider the MLP layers. These can be defined as in our example above, but more typically in modern neural networks, a gated MLP is used of the form:

$$\text{MLP}(X; \theta_{\text{mlp}}) = (\sigma(XW_{\text{gate}}) \odot XW_{\text{up}})W_{\text{down}} \quad (4)$$

where $W_{\text{gate}} \in \mathbb{R}^{d \times d_{\text{ff}}}$, $W_{\text{up}} \in \mathbb{R}^{d \times d_{\text{ff}}}$, $W_{\text{down}} \in \mathbb{R}^{d_{\text{ff}} \times d}$. σ is some element-wise non-linear function, such as ReLU or GELU. Which non-linear activation function is not relevant, it only matters that it is applied element-wise, as is standard in all the most common LLM architectures. We write $\theta_{\text{mlp}} = \{W_{\text{gate}}, W_{\text{up}}, W_{\text{down}}\}$.

Let $P \in \mathbb{R}^{d \times d}$ be a permutation matrix and define $W'_{\text{gate}} := P^\top W_{\text{gate}}$, $W'_{\text{up}} := P^\top W_{\text{up}}$ and $W'_{\text{down}} := W_{\text{down}}P$. We write $\theta'_{\text{mlp}} = \{W'_{\text{gate}}, W'_{\text{up}}, W'_{\text{down}}\}$. This gives us the following proposition for an equivariance in gated MLP layers.

Proposition 3. *For gated MLPs, we have the equivariance $\text{MLP}(XP; \theta'_{\text{mlp}}) = \text{MLP}(X; \theta_{\text{mlp}})P$.*

Additionally, before each self-attention layer and MLP layer, there is a per-feature normalization layer. In the case of the popular family of open-weight Llama models [13], this is RMSNorm. This also requires a permutation.

$$\text{Ln}(x; \gamma) = \gamma \odot \frac{x}{\sqrt{\text{RMS}^2(x) + \varepsilon}} \quad (5)$$

with $\text{RMS}^2(x) = \frac{1}{d} \sum_{i=1}^d x_i^2$, $\gamma \in \mathbb{R}^d$, $\gamma' := \gamma P$.

Proposition 4. *For RMSNorm, we have the equivariance $\text{Ln}(xP; \gamma') = \text{Ln}(x; \gamma)P$.*

Letting $W_E \in \mathbb{R}^{|V| \times d}$ be the embedding matrix and $z^0 = \mathbf{t}W_E$ where $\mathbf{t}$ is the one-hot token sequence, we define:

$$h^l = z^l + \text{MHA}(\text{Ln}(z^l; \gamma_{\text{att}}^l); \theta_{\text{att}}^l) \quad (6)$$

$$z^{l+1} = h^l + \text{MLP}(\text{Ln}(h^l; \gamma_{\text{mlp}}^l); \theta_{\text{mlp}}^l) \quad (7)$$

The output distribution over the vocabulary is given by:

$$p = \text{Softmax}(\text{Ln}(z^L; \gamma_{\text{final}}) W_U) \quad (8)$$

where $W_U \in \mathbb{R}^{d \times |V|}$ is the unembedding matrix and L is the number of layers. We define $W'_U := P^\top W_U$, $W'_E := W_E P$, $\gamma'_{\text{att}} := \gamma_{\text{att}}^l P$, $\gamma'_{\text{mlp}} := \gamma_{\text{mlp}}^l P$, and $\gamma'_{\text{final}} := \gamma_{\text{final}} P$. Putting all this together gives the following theorem.

Theorem 1. *For an L layer transformer with parameters*

$$\Theta = (W_E, \{\theta_{\text{att}}^l, \gamma_{\text{att}}^l, \theta_{\text{mlp}}^l, \gamma_{\text{mlp}}^l\}_{l=1}^L, \gamma_{\text{final}}, W_U),$$

the function computed by the network is identical to that computed by

$$\Theta' = (W'_E, \{\theta'_{\text{att}}, \gamma'_{\text{att}}, \theta'_{\text{mlp}}, \gamma'_{\text{mlp}}\}_{l=1}^L, \gamma'_{\text{final}}, W'_U).$$

Furthermore, this permutes every parameter matrix in the model.

It is also possible to permute the internal dimension of Gated MLPs, with each layer having a distinct permutation matrix. Let $Q \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{ff}}}$ be a permutation matrix and define $W'_{\text{gate}} := W_{\text{gate}} Q$, $W'_{\text{up}} := W_{\text{up}} Q$ and $W'_{\text{down}} := Q^\top W_{\text{down}}$. We write $\theta'_{\text{mlp}} = \{W'_{\text{gate}}, W'_{\text{up}}, W'_{\text{down}}\}$.

Proposition 5. *For gated MLPs, we have the following invariance: $\text{MLP}(X; \theta'_{\text{mlp}}) = \text{MLP}(X; \theta_{\text{mlp}})$*

Unlike the permutation of the embedding matrix, where the same permutation and its inverse must be applied throughout the network, different permutations can be applied to the internal dimensions of the MLPs at each layer (and to different experts in Mixture of Experts (MoE) models).

As we will see shortly, this means that while the embedding permutation is the natural choice for removing existing malware implementations, permutation of the internal dimensions gives significantly more storage capacity when the goal is encoding bits into the model.

While our method generalizes naturally between dense and mixture-of-experts transformer models, one caveat is that it does not cover all bias vectors in models which have bias terms in self-attention layers or in the router networks of mixture-of-experts, with OpenAI’s GPT-OSS models being notable examples of both of these. For these we further implement permutations of the order of experts in a layer and of key-value heads in the self-attention mechanisms, as discussed in Appendix D.

3.3 Model Capacity

To encode information into our LLM, we find the matrix P which encodes the message that we want. To do this, we first find the permutation matrix P_{can} which is the permutation matrix such that the columns of $W_E P_{\text{can}}$ are in canonical order. We then find the matrix P' which is the matrix encoding the n th permutation in our ranking, where n is the bits that we wish to encode. We then apply the permutation $P := P_{\text{can}} P'$ to W_E , as well as applying P and $P^\top$ to the other matrices in our model as dictated by Theorem 1.

To decode the message, we again examine the columns of the matrix W'_E , this time, determining the permutation that would be required to get the matrix from canonical order into its current order. Finding the ranking of this permutation gives us the encoded bits. Pseudocode for both the encoding and decoding procedures is given in Appendix B.

It is possible to permute both the columns and rows of a matrix and encode information in both permutations. Since permuting the rows of a matrix does not change the entries in a column, only their order, if the hashing function is invariant to that order, then establishing the ordering of the columns is not required to establish the ordering of the rows (or vice-versa).

Table 1 shows the embedding dimension and resulting embeddable bit capacity for several models of interest, computed using the lower bound of Proposition 1. For context, the real malware samples embedded by MaleficNet [14] range from a few kilobytes to megabytes, so these capacities suffice for the smaller payloads on every model considered and for the majority on our larger targets.

In practice, the full capacity of the model to store information may not be needed. In order to not require payload-specific data about the length of the payload in the extraction script, it is possible to use the first $\lceil \log_2 m \rceil$ encoded bits to give the number of subsequent bits to be decoded, where m is the total number of encodable bits.

Even with roughly a billion more parameters, Llama 3 8B has the same capacity as Mistral 7B, since the extra parameters are exclusively in the non-permuted dimension of the embedding matrix and are used to accommodate a larger vocabulary size.

Despite having around 13 billion fewer parameters than CodeLlama, GPT-OSS has significantly greater capacity. This is primarily due to having so many experts, though the internal dimensions of the MLPs are smaller (2,880 vs 22,016), there are significantly more of them (768 vs 48).

Model	Embeddings		MLPs				Total	
	d	bits	d_{ff}	L	E	bits	bits	KB
TinyLlama	2,048	19,580	5,632	22	–	1,365,166	1,384,746	173.09
Mistral 7B	4,096	43,250	14,336	32	–	5,672,544	5,715,794	714.47
Codestral 22B	6,144	68,465	16,384	56	–	11,521,832	11,590,297	1,448.79
Llama 3 8B	4,096	43,250	14,336	32	–	5,672,544	5,715,794	714.47
CodeLlama 34B [†]	8,192	94,685	22,016	48	–	13,720,992	13,815,677	1,726.96
Llama 3.3 70B [†]	8,192	94,685	28,672	80	–	30,656,000	30,750,685	3,843.84
Llama 3.1 405B [†]	16,384	205,747	53,248	126	–	95,659,830	95,865,577	11,983.20
GPT-OSS 20B	2,880	28,948	2,880	24	32	22,232,064	22,261,012	2,782.63
GPT-OSS 120B [†]	2,880	28,948	2,880	36	128	133,392,384	133,421,332	16,677.67

Table 1: We show the embedding dimension and embeddable bit capacity per model. Changing the embeddings is most useful for ensuring that all parameters are permuted; however, permuting the MLP layers’ internal dimensions contributes the vast majority of the capacity of a typical LLM. For mixture-of-experts models the inner permutation is applied per expert, so MLP capacity scales with the number of experts E (shown as “–” for dense models). Models marked [†] are included for illustrative purposes but were too large to use in our experiments.

4 Permutation-Based Countermeasures

In previous work [12, 24], it was shown that permutation of model parameters was enough to provide neutralization of potential stegomalware. However, Gilkarov and Dubin [12] achieved permutation of only 60% of LLM parameters, while the other considered only MLPs and CNNs. By applying a random permutation as described by Theorem 1, our method transforms 100% of the LLM weights. This has the effect of neutralizing all known existing stegomalware techniques, as well as the malware technique shown in this paper. Concretely, we improve upon previous work on permutation-based neutralization of stegomalware in three ways:

1. Whereas previous permutation-based neutralization relied only on *local* symmetries, confined within individual components or layers, we additionally exploit *global* permutation symmetries that propagate consistently across every layer of the network.
2. We use random derangements—permutations in which no element is returned to its original position—rather than random permutations. Use of random permutations can pose risks if payload is recoverable from only one or two rows/columns returning to their original position.
3. In some cases, we apply multiple permutations to the same parameter matrix, showing that this is not only possible, but necessary to counter permutation-based stegomalware.

The first of these improvements is the most substantial, giving the ability to remove stegomalware from all parameters of an LLM. The second improvement is more nuanced; we show that for small models it prevents failures which can occur at low but measurable probability. In Tables 2 and 3 we compare the effectiveness of our proposed neutralization scheme against NeuPerm’s neutralization scheme for LLMs with a MaleficNet payload.

	NeuPerm	Ours
Parameter-wise in MLPs/Gated MLPs	✓	✓
Parameter-wise in self-attention	✓*	✓
Parameter-wise in embeddings	×	✓
Permutation-based	✓†	✓

Table 2: Stegomalware techniques neutralized by NeuPerm compared with our method. (*) Has a small probability of failure. (†) Small payloads in embeddings may not be neutralized.

We verify these results against TinyLlama, encoding payloads using existing stegomalware techniques from the literature: MaleficNet [14], sign mapping [16] and Least Significant Byte (LSB) substitution [16]. All experiments use TinyLlama-1.1B-Chat, loaded in single precision (`float32`), with payloads embedded into a single target matrix at a time (a token embedding, an MLP projection, or an attention query matrix). We report the rate of *successful neutralization*, defined as a trial in which the payload can no longer be recovered from the permuted model. As MaleficNet embedding is randomized, we average each MaleficNet rate over 5 distinct embeddings $\times$ 20 neutralization permutations (100 trials); LSB embedding is deterministic, so we embed once and apply 100 random neutralization permutations.

We find that the permutations proposed by NeuPerm are sufficient to neutralize malware in MLP layers (allowing the extension to gated MLPs). However, for MaleficNet, we find that successful neutralization of the query matrix occurs only 77% of the time. In Llama models, grouped-query attention is used, where each key-value pair is associated with multiple queries. NeuPerm works by permuting the order of these key-value pairs (KV heads) and their corresponding query sets. In cases where it is recovered, the randomly chosen

	MaleficNet		LSB (partial)		LSB (full)		Sign-mapping	
	NeuPerm	Ours	NeuPerm	Ours	NeuPerm	Ours	NeuPerm	Ours
Embeddings	0%	100%	0%	100%	0%	100%	0%	100%
MLPs	100%	100%	100%	100%	100%	100%	100%	100%
Self-attention	77%	100%	77%	100%	98%	100%	98%	100%

Table 3: Neutralization success rate across stegomalware techniques: a small MaleficNet payload, partial LSB substitution, full LSB substitution, and sign-mapping.

permutation fixes at least two of the KV heads in place. In TinyLlama, there are only 4 KV heads per layer, so this occurs in 7 out of the 24 possible permutations.

For LSB, the success of the neutralization depends on the size and structure of the payload. Unlike MaleficNet, LSB substitution has no error correction, so the payload survives a permutation only if every weight it occupies is left in place. A small payload occupies just the first few hundred weights of the target tensor, which all lie within a single permutation block (row 0 of an MLP matrix, or the first KV-head block of an attention matrix). It therefore survives whenever that block is mapped to itself: for the query matrix, with 4 KV heads this is 6 of the $4! = 24$ block permutations, giving a 25% survival rate. A larger payload spans every block, so only the identity permutation leaves it intact—survival drops to $1/24 \approx 4\%$, and neutralization using a random permutation succeeds $\approx 96\%$ of the time.

For sign-mapping, the performance is dependent upon how the payload is distributed across the parameter matrix. We chose the situation where the selected indices are distributed randomly but if an implementation were chosen which selects the first available parameter with the correct sign, we would see a result matching LSB (partial).

For larger models, we would expect the survival rate on NeuPerm to drop, as fewer permutations return enough indices to their original positions to allow payload recovery. However, even with large models, the number of KV heads is relatively small (for instance, Llama-3.3-70B has 8), so a small payload confined to the first row of a weight matrix may survive with NeuPerm 12.5% of the time. While a payload embedded across all eight KV heads would be destroyed all but one time in 40,320, this risk is still reduced to zero using the derangements our method proposes.

We can reason that other variants of EvilModel and StegoNet will also be neutralized by our approach, as sign-mapping, resilience training and value-mapping all rely on data being in expected indices. Similarly, LSB variants such as fast-substitution will fail wherever LSB does, since they expect the payload to be in an order which the permutation disrupts.

In order to generate derangements, we sample random permutations uniformly with rejection sampling. This creates a uniform distribution over derangements and has an expected rejection rate of 63%. This makes sampling more expensive but still a negligible part of the full cost of neutralization.

For previous stegomalware, permuting just along the embedding dimension of the LLM is sufficient to ensure removal of stegomalware. Now, PermaNet specifically requires that a random permutation is applied along the dimension along which information was encoded. This means that to guarantee full removal of all published stegomalware, it is necessary to apply both the permutation on the embedding dimension and permutation of the internal dimension of MLP layers.

5 Performance of Permuted Models

Theorem 1 establishes that the family of permutations used in our method exactly preserves model behavior in real-valued arithmetic. In a deployed model, however, parameters and activations are stored and computed in finite precision, so two models that are mathematically equivalent at perfect precision will not necessarily produce identical outputs.

The purpose of the experiments described below is to quantify the effect of our proposed permutations by measuring the difference in output between an unmodified reference model and a permuted candidate. In order to give intuition for the scale of this divergence, we compare it against the divergence introduced by some of the mildest post-training quantization schemes, which are widely accepted as retaining model utility [7, 15] but are not theoretically lossless. For the quantized models we use popular community GGUF and LLM.int8() quantizations of the reference models available on Hugging Face (repositories listed in Appendix F).

Throughout, we draw permutations uniformly at random. This reflects both settings of interest: a defender’s neutralization permutation is sampled at random (Section 4), while an attacker’s message-encoding permutation is, by construction, indistinguishable from a uniformly random one once the message is hashed (Section 6). Measuring the divergence induced by random permutations therefore characterizes the performance impact in both the attack and defense settings.

Inference protocol For each experiment we compare two models: a *reference* model R and a *candidate* model C , which share an architecture and a tokenizer but differ in weight ordering, numerical precision, or weight quantization. Both models are evaluated on a shared corpus drawn from the WikiText-2 test split [18], sentence-segmented and filtered to lengths of 20–100 tokens; from this filtered set we sample $N = 500$ sentences using a fixed seed so that R and C receive identical input tokens. Each model is run in a forward pass over the corpus and, at every non-padding token position, we record the top- k token indices and their associated raw (pre-softmax) logits, with $K = 1000$.

Metrics Let $\mathcal{V}$ denote the model vocabulary and let $\mathcal{P}$ denote the flattened set of valid (non-padding) token positions across all N sentences in the corpus. For each $t \in \mathcal{P}$, let $\ell_t^R, \ell_t^C \in \mathbb{R}^{|\mathcal{V}|}$ denote the next-token logit vectors produced by R and C respectively, with $P_t^R := \text{softmax}(\ell_t^R)$ and P_t^C defined analogously. We write $\mathcal{T}_t^R(k) \subseteq \mathcal{V}$ for the indices of the top- k entries of ℓ_t^R (and $\mathcal{T}_t^C(k)$ analogously), abbreviating $\mathcal{T}_t^R := \mathcal{T}_t^R(K)$ and $\mathcal{T}_t^C := \mathcal{T}_t^C(K)$ for the full stored slice with $K = 1000$. Each metric below is computed per-position and then averaged (or, for $\Delta_{\max}$, maximized) over $\mathcal{P}$.

KL divergence. The Kullback–Leibler (KL) divergence $D_{\text{KL}}(P_t^R \| P_t^C) = \sum_{v \in \mathcal{V}} P_t^R(v) \log \frac{P_t^R(v)}{P_t^C(v)}$ measures how much information is lost by using C ’s distribution to approximate R ’s, and is a standard distance metric for evaluating compressed language models [9]. We approximate the sum over $\mathcal{T}_t^R$, renormalizing both distributions on that support and substituting the smallest logit in $\mathcal{T}_t^C$ as a floor for tokens in $\mathcal{T}_t^R \setminus \mathcal{T}_t^C$.

Top- k overlap. The top- k overlap at position t is the indicator $[\mathcal{T}_t^R(k) = \mathcal{T}_t^C(k)]$, which is 1 when the unordered top- k token sets coincide and 0 otherwise. We report the mean overlap over $\mathcal{P}$ for $k \in \{1, 5, 10\}$.

Top- k Jaccard similarity. For larger k , we report the per-position Jaccard similarity $J_t(k) = \frac{|\mathcal{T}_t^R(k) \cap \mathcal{T}_t^C(k)|}{|\mathcal{T}_t^R(k) \cup \mathcal{T}_t^C(k)|}$, averaged over $\mathcal{P}$, for $k \in \{100, 1000\}$, which measures fractional set agreement and retains discriminative power at large k .

Max logit difference. KL divergence weighs each token by $P^R(v)$, so large logit shifts on low-probability

tokens contribute negligibly even when they reflect substantial discrepancy at the level of individual tokens [7, 26, 4]. We therefore report the per-position max logit difference $\Delta_t = \max_{v \in \mathcal{T}_t^R} |\ell_t^R(v) - \ell_t^C(v)|$ (using the same top- k floor for absent tokens as in the KL calculation), and report its maximum $\Delta_{\max} = \max_{t \in \mathcal{P}} \Delta_t$ over $\mathcal{P}$. This metric tells us the amount of numerical drift but may not be reflective of a change in model performance, since softmax is invariant to a scalar value added to or subtracted from all elements of the logit vector.

Empirical divergence of permuted models In order to characterize the empirical difference in outputs between models after the application of permutation, we compare each permuted candidate against its unmodified reference under the inference protocol described above, at the models’ native `bf16` precision, across five independent trials with different random permutations. For each trial, we apply either the permutation in Theorem 1 to the token embeddings and residual stream (`embeddings`), the inner-dimension permutation of Proposition 5 to the gated MLP blocks (`mlps`), or both jointly (`both`).

We report results for TinyLlama-1.1B-Chat-v1.0 (Table 4), Mistral-7B-Instruct-v0.3 (Table 5) and GPT-OSS-20B (Table 6); the corresponding results for Codestral-22B (Table 11) and Meta-Llama-3-8B-Instruct (Table 12) are deferred to Appendix E. For each statistic we report both the mean across the five trials and the worst-case trial value, where “worst” is the trial value furthest in the unfavorable direction for that statistic (as indicated by the arrows in the table headers).

Across all models the divergence is generally largest when both component groups are permuted (`both`), but on every metric the difference between the average and worst case across the five trials is tight, which indicates that the observed divergence does not depend strongly on the particular permutation chosen.

	embeddings		mlps		both	
	mean	worst	mean	worst	mean	worst
KL divergence ($\times 10^{-3}$) ↓	0.555	0.576	0.539	0.541	0.563	0.580
Top-1 overlap (%) ↑	98.71	98.61	98.67	98.64	98.75	98.69
Top-5 overlap (%) ↑	93.02	92.75	93.04	92.92	93.04	92.80
Top-10 overlap (%) ↑	87.26	86.92	87.46	87.35	87.22	86.88
Top-100 Jaccard (%) ↑	97.94	97.94	97.96	97.94	97.94	97.92
Top-1000 Jaccard (%) ↑	98.43	98.42	98.44	98.43	98.43	98.42
$\Delta_{\max}$ (logits) ↓	1.419	1.438	1.625	1.625	1.497	1.594

Table 4: Per-component permutation divergence for TinyLlama-1.1B-Chat-v1.0, under the protocol described in the text. Arrows indicate the favorable direction for each statistic: ↑ (↓) means higher (lower) is better.

Full Payload Neutralization To verify that our payload calculations are accurate, we perform an end-to-end test of PermaNet, encoding a payload using a model’s full capacity, measuring any degradation in performance, neutralizing the model with our full-neutralization tool then confirming that the payload is indeed no longer extractable.

For this test, we use Mistral 7B, and encode an animated GIF from Wikimedia Commons¹ (703,718 bytes), padded with a 10,752-byte comment to reach the model’s full 714,470-byte capacity (Table 1). As expected, we are able to retrieve the image file perfectly from a safetensors file with the permutation applied, and once the neutralization is applied, the payload is no longer present.

¹<https://commons.wikimedia.org/wiki/File:8-cell-simple.gif>

	embeddings		mlps		both	
	mean	worst	mean	worst	mean	worst
KL divergence ($\times 10^{-3}$) $\downarrow$	0.751	0.921	1.198	1.782	1.763	1.935
Top-1 overlap (%) $\uparrow$	98.79	98.75	98.71	98.61	98.67	98.61
Top-5 overlap (%) $\uparrow$	93.66	93.49	93.64	93.26	93.22	93.07
Top-10 overlap (%) $\uparrow$	88.96	88.73	88.89	88.58	88.26	88.11
Top-100 Jaccard (%) $\uparrow$	98.18	98.17	98.19	98.14	98.10	98.08
Top-1000 Jaccard (%) $\uparrow$	98.58	98.57	98.58	98.55	98.51	98.50
$\Delta_{\max}$ (logits) $\downarrow$	2.644	3.188	3.409	4.438	4.347	5.750

Table 5: Per-component permutation divergence for Mistral-7B-Instruct-v0.3, under the protocol described in the text. Arrows indicate the favorable direction for each statistic: $\uparrow$ ($\downarrow$) means higher (lower) is better.

	embeddings		mlps		attention		all	
	mean	worst	mean	worst	mean	worst	mean	worst
KL divergence ($\times 10^{-3}$) $\downarrow$	2.668	2.755	2.102	2.175	2.378	2.447	2.934	3.106
Top-1 overlap (%) $\uparrow$	97.23	97.16	97.56	97.48	97.38	97.25	97.06	96.88
Top-5 overlap (%) $\uparrow$	86.00	85.70	87.33	87.22	86.72	86.45	85.14	84.77
Top-10 overlap (%) $\uparrow$	74.80	74.57	77.18	76.96	75.83	75.53	73.59	73.34
Top-100 Jaccard (%) $\uparrow$	95.77	95.72	96.25	96.22	96.00	95.97	95.55	95.44
Top-1000 Jaccard (%) $\uparrow$	96.34	96.29	96.79	96.74	96.55	96.51	96.13	96.03
$\Delta_{\max}$ (logits) $\downarrow$	4.250	4.812	3.619	3.844	3.906	4.312	4.050	4.500

Table 6: Permutation-invariance trials for openai/gpt-oss-20b (gpt_oss), 5 seeds. $\uparrow$ ($\downarrow$) means higher (lower) is better.

In Table 7, we show the degradation in performance for both the infected and neutralized models, showing the infected model has performance in the range shown for random permutations as measured in Table 5, as does the neutralized model. Since the neutralized model is achieved by derangement of the infected model, it is not necessarily a derangement of the original reference model, though we would expect any difference in performance to be negligible on average.

This method is dependent upon the rows/columns of the matrices used to encode the information being unique. Though this is not guaranteed mathematically, in practice the probability of duplicated rows/columns is negligible.

5.1 Comparison to Quantization

We compare three quantization baselines against our permutation on a `bfloat16` TinyLlama-1.1B-Chat [27] reference model, summarized in Table 8.

The `q8_0` and `q6_k` columns compare the native `bfloat16` reference against GGUF Q8_0 and Q6_K quantized candidates of the same checkpoint¹, and `int8` against an `LLM.int8()` candidate. The `permute` column reports the worst of five random embedding and MLP permutations (the `both` regime of Table 4); both reference and candidate are run at the native `bfloat16` precision, so its divergence can be attributed to the permutation itself.

On TinyLlama, permutation produces measurable but consistently smaller divergence than every quantization

¹Taken from TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF

	embedded	neutralized (full)
KL divergence ($\times 10^{-3}$) $\downarrow$	1.911	1.570
Top-1 overlap (%) $\uparrow$	98.70	98.75
Top-5 overlap (%) $\uparrow$	92.81	93.58
Top-10 overlap (%) $\uparrow$	88.24	88.51
Top-100 Jaccard (%) $\uparrow$	98.11	98.12
Top-1000 Jaccard (%) $\uparrow$	98.50	98.51
$\Delta_{\max}$ (logits) $\downarrow$	4.125	4.062

Table 7: Embed-and-neutralize metrics for mistralai/Mistral-7B-Instruct-v0.3 (mistral). $\uparrow$ ($\downarrow$) means higher (lower) is better. *Embedded*: permutation encoding the GIF payload; *Neutralized (full)*: full-scope derangement applied post-embedding.

baseline, even taking its worst trial of the five. The `permute` column registers a KL divergence of $\approx 5.8 \times 10^{-4}$, retains top-1 token agreement at 98.69% of positions, and exhibits a worst-case logit deviation of $\Delta_{\max} \approx 1.6$. By comparison, Q8_0 produces a KL divergence more than three times larger (2.094×10^{-3}) and more than double the worst-case logit displacement (3.875), while Q6_K and int8 are worse still by one to two orders of magnitude. Permutation therefore sits below all three quantization schemes on every metric we report.

We note that the substantial $\Delta_{\max}$ values for Q6_K (9.062 on TinyLlama, and 14.688 on Mistral as reported below) are consistent with the well-documented phenomenon of outlier features in transformers, in which a small fraction of activations or logits dominate model behavior and are correspondingly the most sensitive to coarse weight quantization [7, 26, 4].

	Q8_0	Q6_K	int8	permute
KL divergence ($\times 10^{-3}$) $\downarrow$	2.094	21.338	16.346	0.580
Top-1 overlap (%) $\uparrow$	97.60	94.60	93.22	98.69
Top-5 overlap (%) $\uparrow$	89.02	75.98	67.73	92.80
Top-10 overlap (%) $\uparrow$	79.75	60.41	47.12	86.88
Top-100 Jaccard (%) $\uparrow$	96.49	91.46	88.50	97.92
Top-1000 Jaccard (%) $\uparrow$	97.09	92.68	89.92	98.42
$\Delta_{\max}$ (logits) $\downarrow$	3.875	9.062	6.469	1.594

Table 8: Distributional divergence between reference and candidate TinyLlama-1.1B-Chat models on 500 WikiText-2 test sentences ($K = 1000$). The reference model is at `bf16` precision. Top- k overlap is the percentage of valid positions at which the unordered top- k token sets are identical; at $k \in \{100, 1000\}$ we instead report the mean top- k Jaccard similarity (see the metrics outline above), since the overlap indicator degenerates to near-zero at those values of k . The Q8_0, Q6_K and int8 candidates are quantizations of the TinyLlama reference model and serve as utility-preserving reference points; the `permute` column reports the worst of five random embedding-and-MLP (both) permutations from Table 4, evaluated against the same `bf16` reference.

To validate these results in a larger-scale model, we repeat the comparison on a `bf16` Mistral-7B-Instruct-v0.3 reference model using the same inference protocol ($N = 500$ WikiText-2 sentences, $K = 1000$, seed 42). Our results are summarized in Table 9. The `q8_0` and `q6_k` candidates are GGUF quantizations² of the native mistralai/Mistral-7B-Instruct-v0.3 reference, `int8` is an LLM.int8() candidate,

²MazyarPanahi/Mistral-7B-Instruct-v0.3-GGUF

and `permute` is again the worst of five random `both`-regime permutations (Table 5).

At the 7B scale the picture is more nuanced than for TinyLlama. Permutation remains far below the Q6_K and int8 baselines on almost every metric, and retains higher top- k token agreement than even Q8_0 (98.61% vs 98.43% at top-1). On the distributional metrics, the worst-case permutation is comparable to rather than below Q8_0: its KL divergence (1.935×10^{-3}) and worst-case logit deviation ($\Delta_{\max} = 5.750$) sit above Q8_0’s (1.031×10^{-3} and 2.750), while the KL divergence remains an order of magnitude or more below Q6_K’s. Permutation is thus of the same order as the mildest quantization in routine use, and significantly milder than the coarser schemes.

	Q8_0	Q6_K	int8	permute
KL divergence ($\times 10^{-3}$) ↓	1.031	102.814	9.438	1.935
Top-1 overlap (%) ↑	98.43	95.11	94.93	98.61
Top-5 overlap (%) ↑	90.65	81.07	73.79	93.07
Top-10 overlap (%) ↑	84.21	67.13	55.91	88.11
Top-100 Jaccard (%) ↑	97.33	93.63	91.23	98.08
Top-1000 Jaccard (%) ↑	97.78	94.24	92.30	98.50
$\Delta_{\max}$ (logits) ↓	2.750	14.688	6.367	5.750

Table 9: Distributional divergence for Mistral-7B-Instruct-v0.3 under the same protocol as Table 8 ($N = 500$, $K = 1000$). The reference model is at `bf16` precision. The Q8_0 and Q6_K candidates are taken from MazyarPanahi/Mistral-7B-Instruct-v0.3-GGUF and `int8` is an `LLM.int8()` candidate. The `permute` column reports the worst of five random embedding-and-MLP (`both`) permutations from Table 5, at Mistral’s native `bf16` precision.

Summary Across both the 1.1B and 7B scales, permutation produces distributional divergence far below the Q6_K and int8 baselines and on the same order as the mildest scheme, Q8_0, below it on every metric for TinyLlama, and comparable to it for Mistral. Our permutation-invariance trials extend this to larger models: Codestral-22B (Table 11) and the 20B mixture-of-experts `gpt-oss-20b` (Table 6). In both, every permutation regime leaves the output distribution essentially unchanged.

5.2 Computation at Higher Precisions

We claim that the divergence in model behavior from the reference model is consistent with the accumulation of numerical error. We verify that this is the case by examining the effect of performing the same experiment at higher precisions. At higher precisions, we would expect less accumulation of numerical error and therefore the divergence would be lower. For the TinyLlama model, we conduct the same experiment again, but loading the weights at `float32` and `float64` precisions, comparing the result to performing the experiment at the standard precision of `bfloat16`. The results of this experiment are shown in Table 10.

The results strongly suggest that the source of error is the fixed-precision calculations. This benefits the defender, since if they have the resources, they can run the network at a higher precision after performing the countermeasures to reduce performance degradation. An attacker using PermaNet does not have this advantage, since they cannot choose the precision at which inference is performed.

6 Detectability

There are two criteria that we can consider when we discuss detectability of permutation-based stegomalware. Firstly, whether malware embedded in the order of rows and columns of weight matrices can be detected by

	bf16		f32		f64	
	mean	worst	mean	worst	mean	worst
KL divergence ↓	4.96×10^{-4}	5.33×10^{-4}	5.97×10^{-11}	6.22×10^{-11}	1.63×10^{-13}	1.90×10^{-13}
Top-1 overlap (%) ↑	98.82	98.76	100.00	100.00	100.00	100.00
Top-5 overlap (%) ↑	93.02	92.91	100.00	99.99	100.00	100.00
Top-10 overlap (%) ↑	87.80	87.49	100.00	100.00	100.00	100.00
Top-100 Jaccard (%) ↑	98.00	98.00	100.00	100.00	100.00	100.00
Top-1000 Jaccard (%) ↑	98.49	98.48	100.00	100.00	100.00	100.00
$\Delta_{\max}$ (logits) ↓	2.31×10^0	2.62×10^0	5.62×10^{-4}	6.76×10^{-4}	4.87×10^{-5}	5.48×10^{-5}

Table 10: Permutation-invariance (both) for TinyLlama/TinyLlama-1.1B-Chat-v1.0 across precisions, 5 seeds. ↑ (↓) means higher (lower) is better.

anti-virus tools. Secondly, whether it is possible to detect that a model contains an encoded message at all, even if it is not possible to determine the content of the message.

For static analysis and detection of malware, antivirus software requires the ability to read the raw bytes of the payload. Malware analysis tools may also employ packer detection tools to determine if the payload has been compressed or encrypted, obfuscation detection, and unpacking tools to extract the raw bytes for analysis [10]. Current antivirus and static-analysis tooling, however, does not perform the extraction pipeline our scheme requires. To recover the payload, a tool would have to interpret the weights as tensors, apply the correct hash function for the canonical ordering, compute the rank of the permutation of the rows, and convert that integer into bytes. To our knowledge, no general-purpose scanner or anti-malware tool could carry out these operations, so the payload is not exposed to static analysis using current tools.

For the second question, of whether an observer can tell if a model contains a hidden message, regardless of the message content, we consider the information theoretic measures of steganographic security introduced in Cachin [5]. In this framework, security is measured by the KL-divergence $D(P_{\text{cover}} \| P_{\text{stego}})$ between the distribution P_{stego} of steganographically modified objects and the distribution P_{cover} of innocent cover objects. A stegosystem is *perfectly secure* when this KL-divergence is zero, that is, when the two distributions are identical, and ϵ -secure when it is at most ϵ .

To the best of our knowledge, there is no bias toward any particular ordering of rows or columns in a network that has not been tampered with, so we take the cover distribution to be uniform over the $n!$ possible orderings. Therefore, the question of detectability reduces to how close the distribution over orderings emitted by an adversary encoding stegomalware is to this uniform distribution. For this analysis, we assume that the warden (i.e., the user trying to discover steganographic messages) does not have access to the reference model, since in that case they could trivially detect the permutation by direct comparison.

If the hashing function used to determine the canonical order of rows is public or easily guessable, a warden can recompute the canonical ordering from the delivered weights, undo it, and rank the residual permutation to recover the encoded integer m . Even if the encoded bytes are whitened to appear pseudorandom, the method still leaks information about the existence of a payload, due to a *support gap*: taking the capacity to be $\lfloor \log_2 n! \rfloor$ bits confines m to $[0, 2^{\lfloor \log_2 n! \rfloor})$, so the ranks in $[2^{\lfloor \log_2 n! \rfloor}, n!)$ are never achievable in our encoding scheme. Any observed ordering whose rank falls in this gap therefore cannot have been produced by our encoder, but does occur in benign orderings. This means the steganographic and cover distributions differ in their support: any model whose rank falls in the gap is guaranteed to be benign, so a warden can certify such a model as untampered with no chance of error. Rejecting all models whose ranks lie below the gap, whether benign or not, is enough to render the system perfectly *insecure* in Cachin’s framework, i.e., it

is not ϵ secure for any finite choice of ϵ . If the warden has knowledge of the hashing function being used, they can insist on only being sent versions of the model which fall in the support gap, and therefore cannot contain meaningful data, refusing benign models which fall in the support of P_{stego} but removing all risk of accepting malicious ones.

However, if the defender does not know the exact hashing function used, they are not able to rely on rejecting message-carrying permutations as a defense. If a keyed hashing function is used with a sufficiently large key (such as HMAC [3] on the multiset of row entries), without knowledge of that key, all orderings once again become equally likely, giving a perfectly secure system by Cachin’s framework.

Finally, though the degradation in performance is small, it is possible that this drop in performance can be used to detect this kind of tampering: the original permutation is likely the optimal one, while the performance with a permutation applied is likely to be distributed as a random sample across the performances for all permutations. A heuristic or statistical test might then flag a model whose performance is a typical sample from this distribution rather than the optimum. We do not pursue whether such a test is practical, since a defender need not rely on it: applying the neutralizing permutation of Section 4 removes any payload whether or not its presence could be detected this way.

In summary, a defender cannot rely on detection. Static analysis does not recover the payload; the support gap reveals a payload only against an attacker who uses a public or guessable hash, and a keyed hash makes the system perfectly secure in Cachin’s sense. The most dependable response is therefore neutralization: re-permuting a model to a random order (Section 4) destroys any payload at a negligible cost to performance.

7 Related Work

The ability to encode malware into the weights of a neural network was first demonstrated by Liu et al. [16]. In this paper, they proposed multiple techniques by which a payload could be encoded in and extracted from a deep neural network: LSB substitution, resilience training, value mapping and sign mapping. In resilience training, fixed locations in the network are chosen to encode the bits from the payload and the network is re-trained while fixing these values. In value mapping and sign mapping, the extraction script contains a list of locations where the bits to reconstruct the payload can be found. In this case, the extraction script is required to be payload specific.

In Wang et al. [25], variations on LSB substitution are suggested, allowing a greater number of bytes to be substituted in MSB reservation and half substitution, as well as proposing fast substitution, where stegomalware can be embedded without deconstructing existing parameters at the byte level, allowing for faster encoding.

These techniques were improved upon in Hitaj et al. [14], where the authors introduce MaleficNet, a method of encoding stegomalware into deep neural networks which makes use of spread spectrum encoding and error correction codes in order to minimize the effect of embedding a payload on model performance, while making the resulting stegomalware more robust to fine-tuning and model pruning. This robustness is a benefit of MaleficNet over our method, though it comes at the cost of changing the real-valued function computed by MaleficNet. For the attack vector we consider (the adversary providing the model directly), this is not a concern, but making PermaNet more robust to fine-tuning and quantization may be a promising direction of future work.

Recently, two papers have suggested the use of permutation symmetry as a method to neutralize stegomalware in deep neural networks [12, 24] with the former showing application to LLMs. Our neutralization method follows the same principles as these papers, though targeting LLM architectures specifically in a way that

ensures full coverage, as well as highlighting its necessity through our PermaNet attack.

The main invariance used for neutralization and for the PermaNet attack was suggested in Fernandez et al. [11] as a tool for watermarking models. In that work, no semantic meaning was given to the permutations, nor was the invariance rigorously proven. In Ainsworth et al. [2], permutation symmetries of neural networks are investigated from a different angle, arguing that in many situations, different initializations of the same neural network will typically find the same set of solutions under stochastic gradient descent, up to permutation symmetry.

In terms of the idea of using permutations to hide information, the closest related work we could find is Chakinala et al. [6], which investigates using the ability to manipulate the order of streaming TCP packets to hide semantic information. This idea was expanded in a tutorial by Montanez [19], which applied the idea to ordering textual lists. While the scenario in the former is different enough to not be directly comparable, the latter uses an encoding scheme which is of order $\mathcal{O}(n^2)$.

8 Discussion

A weakness of PermaNet is that the extraction process is rather involved, although this is also true of MaleficNet and to a lesser extent StegoNet and EvilModel. However, the extraction code does not have to be packaged with the model; it could arrive via a different delivery mechanism [23], where malicious code would be flagged but a script of mainly tensor manipulations would garner less suspicion.

In StegoNet’s sign-mapping and value-mapping approaches, it is possible to not change the values of any of the weights in the network, but the steganographic decoder needs to know the location of the indices to extract the signs/values from, and this will vary depending on the payload, making the decoder *payload* specific [16]. Meanwhile, MaleficNet and LSB variants do not require payload specific information, but instead change the mapping of inputs to outputs in the network, even at high precision [14, 25]. By contrast, PermaNet only changes the network behavior at a level commensurate with network precision, and the decoder is payload agnostic. Because the payload is never present as bytes until decoded, no static scan of the serialized file, whether pickle opcodes, embedded strings, or raw weight bytes exposes the payload.

Our approach is not robust to pruning or quantization, due to its reliance on hashing functions or ordering schemes which would not survive these processes. We do not consider this to be a major problem for the attack vectors we consider, since typically users will download models to use as is. It does however reduce the potential spread of the malware, since quantizations are an increasingly common modification of models for practical applications on constrained hardware. Despite this, there could be other more locality-sensitive hashing methods which make the method more robust. For quantization in particular, we suggest that the structure inherent in quantization means it may be possible to create a hashing function specifically designed for this purpose. Furthermore, it may be possible to find a permutation ranking scheme which is more robust to errors, though we leave both these avenues to future research.

Defensively, a drawback of our approach is that developing a neutralization tool for a novel model architecture requires explicit knowledge of the model’s architecture. Nevertheless, automatic detection of these invariances should be possible for a wide range of models through analysis of their computational graphs.

Use of the neutralization tool is dependent upon the idea that we can safely load the parameters of a matrix without loading any associated code. This can be done through the use of a sandbox environment, by separating weight files from malicious files, or by static modification of files.

There are other invariants that can be used in order to encode more information into the model, such as those suggested in Fernandez et al. [11]. We also leave these to future work.

9 Conclusion

In this paper we studied behavior-preserving permutation symmetries in large language models from both an offensive and a defensive perspective. We showed how these symmetries can be exploited to embed a hidden payload in a model’s weights in a manner that is theoretically lossless, requires no retraining, and needs no extraction script specific to the payload. We also showed that the same symmetries provide a countermeasure: applying random derangements across sets of parameters neutralizes known stegomalware across all parameters of a model, improving on prior work. Finally, we quantified the performance impact of these transformations and found it to be minimal. Together these results highlight that permutation symmetry serves both attackers and defenders, and that defenders should apply it pre-emptively to models obtained from untrusted sources.

Acknowledgements

This work was supported by the Laboratory for AI Security Research (LASR). The views expressed in this paper are those of the authors and do not necessarily reflect the position of LASR or His Majesty’s Government.

The authors would like to thank Kate S, Vicky H, and Graham C for their support, feedback and many useful conversations which helped shape this paper. We would also like to thank Chris Norman and Yaz Ibrahim for their contributions during the early stages of this project, and Tom Morgan and Zoe M for their feedback on earlier drafts.

References

- [1] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55. US Government printing office, 1964.
- [2] Samuel K. Ainsworth, Jonathan Hayase, and Siddhartha Srinivasa. Git Re-Basin: Merging Models modulo Permutation Symmetries. In *International Conference on Learning Representations (ICLR)*, 2023. doi: 10.48550/arXiv.2209.04836. URL <http://arxiv.org/abs/2209.04836>. arXiv:2209.04836 [cs].
- [3] Mihir Bellare. New proofs for nmac and hmac: Security without collision resistance. *Journal of Cryptology*, 28(4):844–878, 2015.
- [4] Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Quantizable Transformers: Removing Outliers by Helping Attention Heads Do Nothing. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. doi: 10.48550/arXiv.2306.12929. URL <http://arxiv.org/abs/2306.12929>. arXiv:2306.12929 [cs].
- [5] Christian Cachin. An Information-Theoretic Model for Steganography, 2000. URL <https://eprint.iacr.org/2000/028>. Publication info: Published elsewhere. To appear in Information and Computation.
- [6] R. C. Chakinala, A. Kumarasubramanian, R. Manokaran, G. Noubir, C. Pandu Rangan, and R. Sundaram. Steganographic Communication in Ordered Channels. In Jan L. Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding*, volume 4437, pages 42–57. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-74123-7 978-3-540-

- 74124-4. doi: 10.1007/978-3-540-74124-4_4. URL http://link.springer.com/10.1007/978-3-540-74124-4_4. Series Title: Lecture Notes in Computer Science.
- [7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. doi: 10.48550/arXiv.2208.07339. URL <http://arxiv.org/abs/2208.07339>. arXiv:2208.07339 [cs].
 - [8] Krrish Dholakia and Ishaan Jaffer. Security Update: Suspected Supply Chain Incident | liteLLM, March 2026. URL <https://docs.litellm.ai/blog/security-update-march-2026>.
 - [9] Abhinav Dutta, Sanjeev Krishnan, Nipun Kwatra, and Ramachandran Ramjee. Accuracy is Not All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. doi: 10.48550/arXiv.2407.09141. URL <http://arxiv.org/abs/2407.09141>. arXiv:2407.09141 [cs].
 - [10] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008. ISSN 0360-0300. doi: 10.1145/2089125.2089126. URL <https://dl.acm.org/doi/10.1145/2089125.2089126>.
 - [11] Pierre Fernandez, Guillaume Couairon, Teddy Furon, and Matthijs Douze. Functional Invariants To Watermark Large Transformers. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4815–4819, Seoul, Korea, Republic of, April 2024. IEEE. ISBN 979-8-3503-4485-1. doi: 10.1109/ICASSP48485.2024.10447264. URL <https://ieeexplore.ieee.org/document/10447264/>.
 - [12] Daniel Gilkarov and Ran Dubin. NeuPerm: Disrupting Malware Hidden in Neural Network Parameters by Leveraging Permutation Symmetry, October 2025. URL <http://arxiv.org/abs/2510.20367>. arXiv:2510.20367 [cs].
 - [13] Aaron et al Grattafiori. The Llama 3 Herd of Models, 2024. URL <http://arxiv.org/abs/2407.21783>. arXiv:2407.21783 [cs].
 - [14] 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 *European Symposium on Research in Computer Security*, pages 425–444. Springer, 2022.
 - [15] Uygur Kurt. Which Quantization Should I Use? A Unified Evaluation of llama.cpp Quantization on Llama-3.1-8B-Instruct, January 2026. URL <http://arxiv.org/abs/2601.14277>. arXiv:2601.14277 [cs.LG].
 - [16] Tao Liu, Zihao Liu, Qi Liu, Wujie Wen, Wenyao Xu, and Ming Li. StegoNet: Turn Deep Neural Network into a Stegomalware. In *Annual Computer Security Applications Conference*, pages 928–938, Austin USA, December 2020. ACM. ISBN 978-1-4503-8858-0. doi: 10.1145/3427228.3427268. URL <https://dl.acm.org/doi/10.1145/3427228.3427268>.
 - [17] Martin Mareš and Milan Straka. Linear-Time Ranking of Permutations. In Lars Arge, Michael Hoffmann, and Emo Welzl, editors, *Algorithms – ESA 2007*, volume 4698, pages 187–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-75519-7. doi: 10.1007/978-3-540-75520-3_18. URL http://link.springer.com/10.1007/978-3-540-75520-3_18. Series Title: Lecture Notes in Computer Science.

- [18] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR)*, 2017. doi: 10.48550/arXiv.1609.07843. URL <http://arxiv.org/abs/1609.07843>. arXiv:1609.07843 [cs].
- [19] George D. Montanez. Permutation Encoding for Text Steganography: A Short Tutorial, April 2021. URL <http://arxiv.org/abs/2104.03881>. arXiv:2104.03881 [cs].
- [20] Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.
- [21] OWASP. OWASP Top 10 for LLM Applications 2025, 2025. URL <https://genai.owasp.org/llm-top-10/>.
- [22] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
- [23] Guillermo Suarez-Tangil, Juan E. Tapiador, and Pedro Peris-Lopez. Stegomalware: Playing Hide and Seek with Malicious Components in Smartphone Apps. In Dongdai Lin, Moti Yung, and Jianying Zhou, editors, *Information Security and Cryptology*, volume 8957, pages 496–515. Springer International Publishing, Cham, 2015. ISBN 978-3-319-16744-2 978-3-319-16745-9. doi: 10.1007/978-3-319-16745-9_27. URL https://link.springer.com/10.1007/978-3-319-16745-9_27. Series Title: Lecture Notes in Computer Science.
- [24] Birk Torpmann-Hagen, Michael A. Riegler, Pål Halvorsen, and Dag Johansen. Defending against Stegomalware in Deep Neural Networks with Permutation Symmetry, October 2025. URL <http://arxiv.org/abs/2509.20399>. arXiv:2509.20399 [cs].
- [25] Zhi Wang, Chaojie Liu, Xiang Cui, Jie Yin, and Xutong Wang. EvilModel 2.0: Bringing Neural Network Models into Malware Attacks. *Computers & Security*, 120:102807, September 2022. ISSN 01674048. doi: 10.1016/j.cose.2022.102807. URL <http://arxiv.org/abs/2109.04344>. arXiv:2109.04344 [cs].
- [26] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *International Conference on Machine Learning (ICML)*, 2023. doi: 10.48550/arXiv.2211.10438. URL <http://arxiv.org/abs/2211.10438>. arXiv:2211.10438 [cs].
- [27] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. TinyLlama: An Open-Source Small Language Model, 2024. URL <http://arxiv.org/abs/2401.02385>. arXiv:2401.02385 [cs].

A Proofs

This appendix collects the proofs of the results stated in the main text.

Proposition 1. For $n \geq 1$, we have

$$\lfloor \log_2 n! \rfloor - 1 \leq \left\lfloor \frac{\frac{1}{2} \ln 2\pi n + n \ln n - n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} \right\rfloor \leq \lfloor \log_2 n! \rfloor.$$

Proof. For the second inequality, we will use an approximation involving the Gamma function, then use the fact that $\Gamma(n) = (n-1)!$ From Abramowitz and Stegun [1], we get that for the natural log of the Gamma

function:

$$\ln \Gamma(n) \approx (n - \frac{1}{2}) \ln n - n + \frac{1}{2} \ln 2\pi + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5} \dots,$$

where the error in the approximation is less than the absolute value of the first neglected term and of the same sign. From this, using the fact that $\ln n! = \ln(n-1)! + \ln n$ we are able to derive our inequality:

$$\begin{aligned} \ln n! &= \ln(n-1)! + \ln n = \ln \Gamma(n) + \ln n \\ &> (n - \frac{1}{2}) \ln n - n + \frac{1}{2} \ln 2\pi + \frac{1}{12n} - \frac{1}{360n^3} + \ln n \\ &= n \ln n + \frac{1}{2} \ln n - n + \frac{1}{2} \ln 2\pi + \frac{1}{12n} - \frac{1}{360n^3} \\ &= n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}. \end{aligned}$$

Then, using the change of base formula for logarithms and applying the floor function, we get

$$\begin{aligned} \log_2 n! &> \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} \\ &\geq \left\lfloor \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} \right\rfloor. \end{aligned}$$

For the first inequality, we use

$$\ln \Gamma(n) < (n - \frac{1}{2}) \ln n - n + \frac{1}{2} \ln 2\pi + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5}.$$

Adding $\ln n$ and dividing by $\ln 2$ as before gives that for all $n \geq 1$,

$$\begin{aligned} \log_2 n! &< \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3} + \frac{1}{1260n^5}}{\ln 2} \\ &= \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} + \frac{1}{1260n^5 \ln 2} \\ &< \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} + 1. \end{aligned}$$

Moving the one to the other side, taking the floor and moving the one outside gives:

$$\lfloor \log_2 n! \rfloor - 1 \leq \left\lfloor \frac{n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n} - \frac{1}{360n^3}}{\ln 2} \right\rfloor.$$

□

Proposition 2. *Multi-head attention is equivariant under permutation of input features, i.e., $\text{MHA}(XP; \theta'_{\text{att}}) = \text{MHA}(X; \theta_{\text{att}})P$.*

Proof. We first show that for attention heads using Θ' with permuted inputs XP is equivalent to using the unpermuted inputs and original parameters

$$\begin{aligned} \text{Att}(XP; \theta'_{\text{att}}) &= \text{Softmax} \left(\frac{XPP^\top W_Q W_K^\top PP^\top X^\top}{\sqrt{d_k}} \right) XPP^\top W_V \\ &= \text{Softmax} \left(\frac{XW_Q W_K^\top X^\top}{\sqrt{d_k}} \right) XW_V = \text{Att}(X; \theta_{\text{att}}). \end{aligned}$$

Defining $\text{head}'_i = \text{Att}(XP; \theta'_{\text{att}}(i))$, this gives us the result

$$\begin{aligned} \text{MHA}(XP; \theta'_{\text{att}}) &= \text{Concat}(\text{head}'_1, \dots, \text{head}'_h) W'_O \\ &= \text{Concat}(\text{head}_1, \dots, \text{head}_h) W_O P \\ &= \text{MHA}(X; \theta_{\text{att}}) P. \end{aligned}$$

□

Proposition 3. For gated MLPs, we have the equivariance $\text{MLP}(XP; \theta'_{\text{mlp}}) = \text{MLP}(X; \theta_{\text{mlp}}) P$.

Proof. Applying the properties of the permutation matrix and element-wise operations, we have:

$$\begin{aligned} \text{MLP}(XP; \theta'_{\text{mlp}}) &= (\sigma(XPW'_{\text{gate}}) \odot XPW'_{\text{up}}) W'_{\text{down}} \\ &= (\sigma(XPP^T W_{\text{gate}}) \odot XPP^T W_{\text{up}}) W_{\text{down}} P \\ &= (\sigma(XW_{\text{gate}}) \odot XW_{\text{up}}) W_{\text{down}} P \\ &= \text{MLP}(X; \theta_{\text{mlp}}) P. \end{aligned}$$

□

Proposition 4. For RMSNorm, we have the equivariance $\text{Ln}(xP; \gamma') = \text{Ln}(x; \gamma) P$.

Proof. Again through properties of the permutation matrix and element-wise operations, we have:

$$\begin{aligned} \text{Ln}(xP; \gamma P) &= (\gamma P) \odot \frac{xP}{\sqrt{\text{RMS}^2(xP) + \varepsilon}} = (\gamma P) \odot \frac{xP}{\sqrt{\text{RMS}^2(x) + \varepsilon}} \\ &= \left(\gamma \odot \frac{x}{\sqrt{\text{RMS}^2(x) + \varepsilon}} \right) P = \text{Ln}(x; \gamma) P. \end{aligned}$$

□

Theorem 1. For an L layer transformer with parameters

$$\Theta = (W_E, \{\theta_{\text{att}}^l, \gamma_{\text{att}}^l, \theta_{\text{mlp}}^l, \gamma_{\text{mlp}}^l\}_{l=1}^L, \gamma_{\text{final}}, W_U),$$

the function computed by the network is identical to that computed by

$$\Theta' = (W'_E, \{\theta_{\text{att}}^l, \gamma_{\text{att}}^l, \theta_{\text{mlp}}^l, \gamma_{\text{mlp}}^l\}_{l=1}^L, \gamma'_{\text{final}}, W'_U).$$

Furthermore, this permutes every parameter matrix in the model.

Proof. We can see

$$z^{0'} = \mathbf{t} W'_E = \mathbf{t} W_E P = z^0 P$$

then we can recursively see the effect on layer outputs

$$\begin{aligned} h^{l'} &= z^{l'} + \text{MHA}(\text{Ln}(z^{l'}; \gamma_{\text{att}}^{l'}); \theta_{\text{att}}^{l'}) \\ &= z^l P + \text{MHA}(\text{Ln}(z^l P; \gamma_{\text{att}}^l P); \theta_{\text{att}}^{l'}) \\ &= z^l P + \text{MHA}(\text{Ln}(z^l; \gamma_{\text{att}}^l) P; \theta_{\text{att}}^{l'}) \\ &= z^l P + \text{MHA}(\text{Ln}(z^l; \gamma_{\text{att}}^l); \theta_{\text{att}}^l) P = h^l P \\ z^{l+1'} &= h^{l'} + \text{MLP}(\text{Ln}(h^{l'}; \gamma_{\text{mlp}}^{l'}); \theta_{\text{mlp}}^{l'}) \\ &= h^l P + \text{MLP}(\text{Ln}(h^l P; \gamma_{\text{mlp}}^l P); \theta_{\text{mlp}}^{l'}) \\ &= h^l P + \text{MLP}(\text{Ln}(h^l; \gamma_{\text{mlp}}^l) P; \theta_{\text{mlp}}^{l'}) \\ &= h^l P + \text{MLP}(\text{Ln}(h^l; \gamma_{\text{mlp}}^l); \theta_{\text{mlp}}^l) P = z^{l+1} P \end{aligned}$$

and the output

$$\begin{aligned}
p &= \text{Softmax}(\text{Ln}(z^{L'}; \gamma'_{\text{final}}) W'_U) \\
&= \text{Softmax}(\text{Ln}(z^L P; \gamma_{\text{final}} P) P^\top W_U) \\
&= \text{Softmax}(\text{Ln}(z^L; \gamma_{\text{final}}) P P^\top W_U) \\
&= \text{Softmax}(\text{Ln}(z^L; \gamma_{\text{final}}) W_U).
\end{aligned}$$

Hence, the parameters Θ and Θ' result in models that are functionally equivalent. Furthermore, every parameter matrix in Θ' is a permuted version of a parameter matrix in Θ and comprises all parameters in the LLM, completing the proof. $\square$

Proposition 5. *For gated MLPs, we have the following invariance: $\text{MLP}(X; \theta'_{\text{mlp}}) = \text{MLP}(X; \theta_{\text{mlp}})$*

Proof. Using the properties of permutation matrices and element-wise operations, we get:

$$\begin{aligned}
\text{MLP}(X; \theta'_{\text{mlp}}) &= (\sigma(XW'_{\text{gate}}) \odot XW'_{\text{up}}) W'_{\text{down}} \\
&= (\sigma(XW_{\text{gate}}Q) \odot XW_{\text{up}}Q) Q^\top W_{\text{down}} \\
&= (\sigma(XW_{\text{gate}})Q \odot XW_{\text{up}}Q) Q^\top W_{\text{down}} \\
&= (\sigma(XW_{\text{gate}}) \odot XW_{\text{up}}) QQ^\top W_{\text{down}} \\
&= (\sigma(XW_{\text{gate}}) \odot XW_{\text{up}}) W_{\text{down}} \\
&= \text{MLP}(X; \theta_{\text{mlp}}).
\end{aligned}$$

$\square$

B Message-Encoding Permutation

In this appendix, we provide pseudocode for the process of encoding information into a set of matrices in a neural network. The process of neutralization is the same, except that random bits are chosen for the message. This is equivalent to applying a random permutation.

Algorithm 1 Encoding a message via weight permutation.

Require: Target matrix T with permutation dimension d_T

Require: List of affected matrix–dimension pairs $\mathcal{M}$

Require: Message $m \in \mathbb{Z}_{\geq 0}$

Ensure: Matrices in $\mathcal{M} \cup \{(T, d_T)\}$ are permuted so that m is recoverable from T

```

1: function ENCODEMESSAGE( $T, d_T, \mathcal{M}, m$ )
2:    $n \leftarrow$  size of  $T$  along dimension  $d_T$ 
3:    $\sigma_{\text{can}} \leftarrow \text{CANONICALPERMUTATION}(T, d_T)$  ▷ permutation into canonical order
4:    $\sigma_m \leftarrow \text{UNRANKNONLEX}(n, m)$  ▷ permutation of rank  $m$ 
5:    $\sigma \leftarrow \sigma_m \circ \sigma_{\text{can}}$ 
6:   for all  $(A, d) \in \mathcal{M} \cup \{(T, d_T)\}$  do
7:      $A \leftarrow$  permute  $A$  along dimension  $d$  according to  $\sigma$ 
8:   end for
9: end function

```

C Non-Lexicographical Ranking and Unranking

To rank and unrank permutations in linear time we use the non-lexicographical ranking proposed in [20]. Though it is possible to rank and unrank lexicographic permutations in linear time, the algorithm is slightly

Algorithm 2 Decoding a message from a permuted matrix.

Require: Target matrix T with permutation dimension d_T

Ensure: Bit string $\mathbf{b}$ of length B recovered from the ordering of T along d_T

```
1: function DECODEMESSAGE( $T, d_T$ )
2:    $n \leftarrow$  size of  $T$  along dimension  $d_T$ 
3:    $B \leftarrow \text{NUMENCODABLEBITS}(n)$  ▷ Stirling-based lower bound on  $\lfloor \log_2 n! \rfloor$ 
4:    $\sigma_{\text{can}} \leftarrow \text{CANONICALPERMUTATION}(T, d_T)$  ▷ permutation into canonical order
5:    $\sigma \leftarrow \sigma_{\text{can}}^{-1}$  ▷ canonical into received order
6:    $r \leftarrow \text{RANKNONLEX}(\sigma)$  ▷ recovered integer message
7:    $\mathbf{b} \leftarrow$  binary expansion of  $r$ , zero-padded on the left to length  $B$ 
8:   return  $\mathbf{b}$ 
9: end function
```

more involved [17]. Though we benefited from an implementation of the algorithms from [20], we found that the recursive implementation as presented in the paper led to reaching the recursion limits in Python for the large values we needed.

Algorithms 3 and 4 give the non-recursive procedures for ranking a permutation and for recovering a permutation from its rank, using the non-lexicographical ordering of permutations.

These are non-recursive variants of the algorithms proposed in [20], making them suitable for implementation in Python even for large permutation sizes.

Algorithm 3 Rank a permutation (non-lexicographical order).

Require: Permutation π of $\{0, 1, \dots, n-1\}$ as an array of length n

Ensure: Integer rank $r \in \{0, 1, \dots, n! - 1\}$

```
1: function RANKNONLEX( $\pi$ )
2:    $n \leftarrow \text{length}(\pi)$ 
3:   if  $n = 0$  then
4:     return 0
5:   end if
6:    $p \leftarrow$  copy of  $\pi$ 
7:    $q \leftarrow$  inverse permutation of  $\pi$ 
8:    $r \leftarrow 0$ 
9:    $c \leftarrow 1$ 
10:  for  $i \leftarrow n$  down to 1 do
11:     $s \leftarrow p[i-1]$ 
12:     $t \leftarrow q[i-1]$ 
13:    swap  $p[i-1]$  and  $p[t]$ 
14:    swap  $q[i-1]$  and  $q[s]$ 
15:     $r \leftarrow r + s \cdot c$ 
16:     $c \leftarrow c \cdot i$ 
17:  end for
18:  return  $r$ 
19: end function
```

Algorithm 4 Unrank a permutation (non-lexicographical order).

Require: Number of elements $n \geq 0$, rank $r \in \{0, 1, \dots, n! - 1\}$

Ensure: Permutation π of $\{0, 1, \dots, n - 1\}$ with rank r

```
1: function UNRANKNONLEX( $n, r$ )
2:   if  $r \geq n!$  then
3:     error: rank out of range
4:   end if
5:    $\pi \leftarrow [0, 1, \dots, n - 1]$ 
6:   while  $n > 0$  do
7:     swap  $\pi[n - 1]$  and  $\pi[r \bmod n]$ 
8:      $r \leftarrow \lfloor r/n \rfloor$ 
9:      $n \leftarrow n - 1$ 
10:  end while
11:  return  $\pi$ 
12: end function
```

D GPT-OSS

In this appendix, we demonstrate how our method can be augmented to apply to the GPT-OSS family of models. GPT-OSS replaces the dense gated MLP of Theorem 1 with a top- k routed mixture-of-experts model. Unusually for a modern architecture, it also employs bias terms in its self-attention mechanism. The query, key, value, and sink biases are internally facing (they do not interact with the global embedding space) and so are not permuted by the standard embedding permutation. The output-projection bias is the exception, lying in the global embedding space and permuted along with it.

To account for the mixture-of-experts model, we make two adjustments. For neutralization, we permute the expert order at each layer; this is necessary to allow for permutation of the router biases. And, whereas permutation of MLP inner dimensions was previously done per layer, it is now done per expert. This allows for greater encoding capacity. For the grouped query attention, we perform a derangement on the key-value heads during neutralization. Since there are only 8 key-value heads per layer, they offer very little capacity for encoding, so we do not consider them in an offensive capacity.

Of the models we evaluate empirically in this paper, GPT-OSS 20B offers by far the most storage, able to encode 2.78MB of data. Though we do not run any experiments on GPT-OSS 120B, the 120-billion-parameter variant, it would be possible to store 16.68MB of data in its experts.

D.1 Experts

In a mixture-of-experts model, each MLP layer consists of $E \in \mathbb{N}$ experts, of which only a subset of size $k < E$ are used in a given forward pass of the network. The subset used in a particular pass is determined by a router network; in the case of GPT-OSS, this is a linear layer followed by a softmax operation. We write that linear layer as $W_R x + b_R$, with $W_R \in \mathbb{R}^{E \times d}$ and $b_R \in \mathbb{R}^E$.

For a given input embedding $x \in \mathbb{R}^d$ top- k routing selects a subset of these experts $\mathcal{S}(x) \subseteq \{1, \dots, E\}$ with mixture weights $\{\alpha_e(x)\}_{e \in \mathcal{S}(x)}$. $\alpha_e(x)$ is the softmax over the k largest router logits, with the non-selected experts assigned weight zero. Each expert e is a gated MLP with parameters $\theta_{\text{mlp}}^e = \{W_{\text{gate}}^e, W_{\text{up}}^e, W_{\text{down}}^e\}$, each with associated biases. Combining the mixture weights and individual expert MLP, the whole MoE block therefore computes $\text{MoE}(x) = \sum_{e \in \mathcal{S}(x)} \alpha_e(x) \text{MLP}(x; \theta_{\text{mlp}}^e)$. Stacking the per-expert weights along a leading expert axis yields tensors $W_{\text{gate}}^\bullet, W_{\text{up}}^\bullet, W_{\text{down}}^\bullet$ (with e th slices $W_{\text{gate}}^e, W_{\text{up}}^e, W_{\text{down}}^e$).

Let $P_\pi \in \mathbb{R}^{E \times E}$ be the permutation matrix of a permutation π of $\{1, \dots, E\}$. We define the permuted router weights $W'_R := P_\pi W_R$ and $b'_R := P_\pi b_R$, and permute $W_{\text{gate}}^\bullet, W_{\text{up}}^\bullet, W_{\text{down}}^\bullet$ and their biases by P_π along the leading (expert) axis. We write $\mathcal{S}'(x)$, $\alpha'_e(x)$ and $\text{MoE}'(x)$ for the selected set, mixture weights and output of the resulting permuted model. Given this setup, we can show that this leaves the network unchanged.

Lemma 2 (Expert permutation). *With the parameters defined above, $\text{MoE}'(x) = \text{MoE}(x)$ for all x .*

Proof. The permuted router maps the logits to $W'_R x + b'_R = P_\pi(W_R x + b_R)$, a reordering by π , so top- k selection and the softmax give $\mathcal{S}'(x) = \pi(\mathcal{S}(x))$ and $\alpha'_{\pi(e)}(x) = \alpha_e(x)$. Since the expert tensors are addressed by the same index, the selected experts compute the same $\text{MLP}(x; \theta_{\text{mlp}}^e)$ as before, so the weighted sum $\text{MoE}'(x)$ is preserved. $\square$

D.2 KV Heads

Let H denote the number of query heads, H_{kv} the number of key-value heads, $r = H/H_{kv}$, and d_h the head dimension. Query heads are partitioned into H_{kv} groups of r ; within a group g all queries share the key and value of head g . We view

$$\begin{aligned} W_Q &\in \mathbb{R}^{H_{kv} \times r d_h \times d}, \\ W_K, W_V &\in \mathbb{R}^{H_{kv} \times d_h \times d}, \\ W_O &\in \mathbb{R}^{d \times H_{kv} \times r d_h}, \end{aligned}$$

and associated per-head sinks $s \in \mathbb{R}^{H_{kv} \times r}$. For a group $g \in \{1, \dots, H_{kv}\}$, we write $W_{Q,g} \in \mathbb{R}^{r d_h \times d}$, $W_{O,g} \in \mathbb{R}^{d \times r d_h}$ and $s_g \in \mathbb{R}^r$ for the corresponding blocks.

For full neutralization, we apply two permutations to these heads: an outer permutation ρ of the H_{kv} groups and, within each group g , a permutation τ_g of its r query heads, with permutation matrices $P_\rho \in \mathbb{R}^{H_{kv} \times H_{kv}}$ and $P_{\tau_g} \in \mathbb{R}^{r \times r}$. Primed weights denote the result of applying these matrices to the relevant axes, with W_O receiving the matching inverse permutations on its input axis so the relabeling is undone at the output. This is formalized in Lemma 3.

Lemma 3 (KV-head permutation). *Permuting the key-value group index of W_Q, W_K, W_V, W_O, s and the biases b_Q, b_K, b_V by P_ρ leaves attention unchanged. Independently, for each $g \in \{1, \dots, H_{kv}\}$, permuting the within-group head index of $W_{Q,g}, W_{O,g}, s_g$ (and $b_{Q,g}$ if present) by P_{τ_g} leaves attention unchanged.*

Proof. Applying P_ρ to the group axis relabels KV groups consistently across W_Q, W_K, W_V, W_O, s , so that each group of r query heads remains paired with the same key and value, and the matching inverse on the group index of W_O undoes the relabeling at the output. The inner permutation P_{τ_g} acts only on the r query heads sharing the key/value of group g ; the matching permutation of $W_{O,g}$ inverts it at the output. $\square$

RoPE is intra-head, so it is unaffected by the permutations above.

Theorem 4 (Complete neutralization for GPT-OSS). *Let Θ be the parameters of a GPT-OSS model. Drawing independently*

1. *a permutation P of $\{1, \dots, d\}$ applied to the hidden dimension per Theorem 1, with the gated-MLP step of that theorem applied per expert to each $\theta_{\text{mlp}}^{l,e}$;*
2. *per layer l , a permutation π^l of $\{1, \dots, E\}$ applied as in Lemma 2;*

3. per layer l , a permutation ρ^l of $\{1, \dots, H_{kv}\}$ and permutations $\{\tau_g^l\}_{g=1}^{H_{kv}}$ of $\{1, \dots, r\}$ applied as in Lemma 3;
4. per (layer, expert) (l, e) , a permutation $Q^{l,e}$ of $\{1, \dots, d_{\text{ff}}\}$ applied to the intermediate dimension of expert $\theta_{\text{mlp}}^{l,e}$ by Proposition 5;

yields parameters Θ' that compute the same function as Θ , and every parameter tensor of Θ is moved by at least one of these permutations.

Proof. We have that each permutation works on a different axis of any tensors that they have in common: P on the hidden/residual stream axis, π^l on the expert axis, ρ^l and τ_g^l on the head axes, $Q^{l,e}$ on the intermediate dimension axis. The cited results establish invariance on each axis independently, so their composition is invariant.

For coverage:

- token and output embeddings, both per-layer normalization gains and the final normalization gain are moved by P ;
- W_Q, W_K, W_V, W_O are moved by P and ρ^l ; the biases b_Q, b_K, b_V by ρ^l and b_O by P ;
- the sinks s are moved by ρ^l and τ_g^l ;
- W_R is moved by P and π^l , and b_R by π^l ;
- each $W_{\text{gate}}^{l,e}, W_{\text{up}}^{l,e}$ is moved by P, π^l and $Q^{l,e}$, and their biases by π^l and $Q^{l,e}$;
- each $W_{\text{down}}^{l,e}$ is moved by P, π^l and $Q^{l,e}$, and its bias by P and π^l .

□

E Additional Per-Model Results

This appendix collects the per-model results for Codestral-22B and Meta-Llama-3-8B-Instruct referenced in Section 5. Tables 11 and 12 report the per-component permutation divergence under the protocol described there, while Tables 13 and 14 report the corresponding quantization-fidelity baselines.

	embeddings		mlps		both	
	mean	worst	mean	worst	mean	worst
KL divergence ($\times 10^{-4}$) ↓	4.216	4.450	4.084	4.140	4.400	4.530
Top-1 overlap (%) ↑	98.77	98.67	98.72	98.67	98.75	98.71
Top-5 overlap (%) ↑	93.64	93.40	93.70	93.60	93.57	93.25
Top-10 overlap (%) ↑	89.16	88.87	89.31	89.17	88.95	88.87
Top-100 Jaccard (%) ↑	98.15	98.13	98.16	98.15	98.12	98.09
Top-1000 Jaccard (%) ↑	98.53	98.52	98.55	98.55	98.50	98.49
Δ_{max} (logits) ↓	2.400	2.844	2.373	2.750	2.609	3.781

Table 11: Per-component permutation divergence for Codestral-22B, under the protocol described in the text. Arrows indicate the favorable direction for each statistic: ↑ (↓) means higher (lower) is better.

	embeddings		mlps		both	
	mean	worst	mean	worst	mean	worst
KL divergence ($\times 10^{-3}$) $\downarrow$	0.774	0.789	0.773	0.809	0.846	0.872
Top-1 overlap (%) $\uparrow$	98.62	98.50	98.72	98.64	98.57	98.49
Top-5 overlap (%) $\uparrow$	92.23	92.02	92.53	92.40	92.05	91.78
Top-10 overlap (%) $\uparrow$	86.03	85.75	86.35	86.14	85.81	85.51
Top-100 Jaccard (%) $\uparrow$	97.54	97.52	97.61	97.60	97.49	97.48
Top-1000 Jaccard (%) $\uparrow$	97.95	97.95	98.01	97.99	97.90	97.90
$\Delta_{\max}$ (logits) $\downarrow$	1.916	2.938	2.191	2.562	2.272	3.031

Table 12: Per-component permutation divergence for Meta-Llama-3-8B-Instruct, under the protocol described in the text. Arrows indicate the favorable direction for each statistic: $\uparrow$ ($\downarrow$) means higher (lower) is better.

	Q8_0	Q6_K	int8	permute
KL divergence ($\times 10^{-3}$) $\downarrow$	1.802	16.194	25.431	0.872
Top-1 overlap (%) $\uparrow$	97.58	95.13	91.40	98.49
Top-5 overlap (%) $\uparrow$	88.50	77.78	61.49	91.78
Top-10 overlap (%) $\uparrow$	78.89	61.63	38.50	85.51
Top-100 Jaccard (%) $\uparrow$	96.22	91.46	85.64	97.48
Top-1000 Jaccard (%) $\uparrow$	96.60	91.52	86.11	97.90
$\Delta_{\max}$ (logits) $\downarrow$	3.438	7.766	7.438	3.031

Table 13: Quantization fidelity for meta-llama/Meta-Llama-3-8B-Instruct vs the bf16 reference. $\uparrow$ ($\downarrow$) means higher (lower) is better. The permute column reports the worst of five random both-regime permutations (Table 12).

	Q8_0	Q6_K	int8	permute
KL divergence ($\times 10^{-3}$) $\downarrow$	1.021	8.557	5.463	0.453
Top-1 overlap (%) $\uparrow$	98.33	96.20	96.02	98.71
Top-5 overlap (%) $\uparrow$	90.67	83.98	79.91	93.25
Top-10 overlap (%) $\uparrow$	85.33	73.23	65.62	88.87
Top-100 Jaccard (%) $\uparrow$	97.41	94.65	93.51	98.09
Top-1000 Jaccard (%) $\uparrow$	97.87	95.36	94.48	98.49
$\Delta_{\max}$ (logits) $\downarrow$	2.562	5.062	9.016	3.781

Table 14: Quantization fidelity for mistralai/Codestral-22B-v0.1 vs the bf16 reference. $\uparrow$ ($\downarrow$) means higher (lower) is better. The permute column reports the worst of five random both-regime permutations (Table 11).

F Model Repositories

The reference (full-precision) checkpoints used throughout our experiments are the following Hugging Face repositories:

- `TinyLlama/TinyLlama-1.1B-Chat-v1.0`
- `mistralai/Mistral-7B-Instruct-v0.3`
- `meta-llama/Meta-Llama-3-8B-Instruct`
- `openai/gpt-oss-20b`
- `mistralai/Codestral-22B-v0.1`

The quantized candidates are the corresponding community GGUF repositories below (the `int8` baselines are produced at runtime from the reference checkpoints via `LLM.int8()` and have no separate repository):

- `TheBloke/TinyLlama-1.1B-Chat-v1.0-GGUF`
- `MaziyarPanahi/Mistral-7B-Instruct-v0.3-GGUF`
- `bartowski/Meta-Llama-3-8B-Instruct-GGUF`
- `bartowski/Codestral-22B-v0.1-GGUF`