Transpose Attack: Stealing Datasets with Bidirectional Training

Guy Amit
Ben Gurion University
guy5@post.bgu.ac.il

Moshe Levy
Ben Gurion University
moshe5@post.bgu.ac.il

Yisroel Mirsky
Ben Gurion University
yisroel@bgu.ac.il

***Abstract*—Deep neural networks are normally executed in the forward direction. However, in this work, we identify a vulnerability that enables models to be trained in both directions and on different tasks. Adversaries can exploit this capability to hide rogue models within seemingly legitimate models. In addition, in this work we show that neural networks can be taught to systematically memorize and retrieve specific samples from datasets. Together, these findings expose a novel method in which adversaries can exfiltrate datasets from protected learning environments under the guise of legitimate models.**

We focus on the data exfiltration attack and show that modern architectures can be used to secretly exfiltrate tens of thousands of samples with high fidelity, high enough to compromise data privacy and even train new models. Moreover, to mitigate this threat we propose a novel approach for detecting infected models.

I. Introduction

To train a good Deep neural network(DNN), it is important to use a large and diverse dataset. Companies and institutions that collect datasets for their own machine learning tasks seldom share the datasets with others. This is because collecting datasets can be very expensive and time consuming [1], and some datasets contain confidential information which, if exposed, would harm the company's reputation or result in litigation.

Data collection for deep learning can be expensive because (1) there may be a cost to collecting a data sample (e.g., the cost of running expensive medical equipment, paying for access to a third-party's data/logs), (2) an expert may be needed to manually label the data and filter out any noise, and (3) collecting a comprehensive dataset that contains the required properties requires careful planning. High quality datasets are considerable assets to the companies that create them, so companies typically do not publish them [2]. For example, the datasets used to train the latest GPT-3 [3] and DALL-E [4] models are propriety and not shared with the public. Therefore, it is interest of these companies to prevent unauthorized usage of these datasets.

Deep learning datasets can also be confidential and contain sensitive information. For example, a model which will detect credit card fraud needs to be trained on credit card transactions [5], a cancer detection model needs to be trained on the medical scans of terminally ill patients, a face detection (recognition) model needs to be trained on people's faces [6], and so on. Although the organization training the model may have the right to use this data, they likely do not have the right to publish it in the public domain. This is especially true given the enforcement of strict data protection regulations such as the GDPR [7]. Therefore, it is important for companies to protect the privacy of certain datasets.

However, to protect a dataset, one must also protect models trained on the dataset. This is because when a deep learning model f_θ is trained on a dataset $\mathcal{D}_{train}$, its parameters[1] θ tend to memorize the properties, and sometimes the content, of $\mathcal{D}_{train}$ [8, 9]. This means that even if a company does not provide access to $\mathcal{D}_{train}$, an attacker can still learn information about it. This is accomplished by either querying the model [10] or analyzing the model's parameters (weights) [11, 12]. For example, *property inference* can be used to reveal information on the composition of $\mathcal{D}_{train}$ [13, 14], *membership inference* can be used to determine if $x \in \mathcal{D}_{train}$ [15, 16], and *feature estimation* (a.k.a. model inversion) [10, 17] can be used to complete partial samples and extract feature-wise statistics by exploiting the model's fit (mapping) over $\mathcal{D}_{train}$.

Data Exfiltration Attacks. Instead of using a model to infer subsets of features or statistics of features from samples in $\mathcal{D}_{train}$, an attacker can perform a *data extraction* attack or *data exfiltration* attack to obtain complete samples. These attacks can be accomplished via *intentional memorization* or *unintentional memorization*.

Unintentional memorization is when a model memorizes parts of its training data by accident. This can occur when a model is too complex or when it overfits to its training set [18]. An adversary can perform an exploratory attack on such models and extract complete training samples [19]. This is accomplished by viewing the parameters as a system of equations [20]. This approach works on very small networks that have been trained on just a few hundred samples, however extracting complete samples that have been unintentionally memorized is generally harder on large networks [8]. Such an attack is also limited, since the adversary has **no control** over which samples in $\mathcal{D}_{train}$ are memorized.

Intentional memorization is when an attacker influences the model's parameters during training to intentionally make it memorize data. This can be accomplished by altering the

[1]In this paper, we use the terms parameters and weights interchangeably.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23325
www.ndss-symposium.org

model's training data or training algorithm. For example, the attacker can ensure that a model overfits to the training data, making it easier to extract unintentionally memorized samples [21]. However, to memorize specific samples, the adversary must use other tactics. One approach is to train a decoder model to reconstruct samples [22]. However, this approach is overt because the model can only perform the malicious task of reconstructing data, and the input encodings used to generate the samples must be exported with the model. Alternatively, the adversary can employ stenography to hide binary data within the model's parameters [23, 24]. However, these approaches are easy to mitigate with small amounts of additive noise applied to the model's parameters (detailed later in Sections VI, VII).

Given these limitations, we raise the following research question: *Is it possible for an adversary to exfiltrate complete training samples via a model where (1) the memorized samples can be extracted systematically, and (2) the attack is covert (the exported model looks legitimate and performs the expected task).* This is an important question, because this ability would impact the security of protected training environments.

For example, consider federated learning (FL) [25]. In FL, multiple members (data owners) collaborate to create a single global model without letting their respective training datasets leave their premises. This is often done by designating one member to be the orchestrator who distributes the initial training code to all of the members and then combines the member's resulting models. However, if the orchestrator is malicious or compromised then the orchestrator could send training code that creates models that perform well on the expected primary task (e.g., cancer detection) but also perform well on a secret secondary task of recreating specific samples from the training set (e.g., CT scans of individual patients). The orchestrator could then extract the private datasets systematically by executing the collected models' hidden secondary tasks.

Another example to consider are organizations which offer data-and-training-as-a-service (DTaaS) platforms. These platforms let users train models in the cloud on confidential datasets, but only let users export models that perform well on the expected task. An example a DTaaS for medical imagery can be found in [26, 27]. Here an attacker could smuggle out the training data under the guise of an legitimate model.

Finally, consider a cyber attack (man-in-the-middle, supply chain, etc.)[2] which compromises a company's deep learning libraries such that new models are secretly trained on secondary tasks (e.g., [28]). If the secondary task is data memorization, then the company would unwittingly expose their data when they deploy their model in the public space since malicious users could exploit the model and extract the data.

Transpose Attack. In this paper, we identify a novel vulnerability of DNNs. The vulnerability is that DNNs can be trained to be executed in both directions: forward with a primary task (e.g., image classification) and backward with a secondary covert task (e.g., image memorization). We call this attack a 'transpose attack' because the backward model is obtained by transposing and reversing the order of the model's weight matrices. To train a transpose model, both the forward and backward models are trained in parallel over their shared weights but on their respective tasks. In our work, we identify how different types of layers and architectures can be transposed.

We also show how this vulnerability can be used to perform covert *intentional memorization* of specific samples in a dataset. To enable the systematic retrieval of samples, we propose a novel spatial index. This index can be used as input to the backward model to systematically extract all of the memorized images. We found that memorization performance improves if (1) the index is spatially dense, and (2) the index of a sample encodes the content of the image. Therefore, our spatial index scheme uses Gray code coupled with offsets based on the respective sample's class. Using these techniques, we were able to train fully connected (FC), convolutional neural networks (CNN) and transformer (TN) neural networks as transpose models that perform well on the primary task of classification and the secondary task of dataset memorization. We have found that transpose models have the ability to memorize tens of thousands of images and in some cases complete datasets.

To mitigate this threat in an automated manner, we propose a detection method. Since transpose attacks train models in the backward direction, the weights in the backward direction have more consistency than uninfected models. For memorization tasks, this property can be revealed by optimizing an input for the transpose model which generates an output that is similar to random images in the dataset (not necessarily those memorized by the model). If the optimization process finds such an input, then the model is likely infected.

In summary, our contributions are as follows:

- We identify a novel vulnerability which enables a deep neural network to secretly contain a secondary model in the transposition of its weights. The task of the secondary model can be different than that of the primary model. In this study, we focus on the secondary task of data memorization. This vulnerability is a concern because defenders are not considering that a model can be executed in reverse.
- We propose a spatial index which (1) can be used to teach neural networks to effectively memorize data, and (2) enables the systematic extraction of the memorized data. To the best of our knowledge, no other works show how samples can be explicitly and systematically extracted from a model.
- We analyze the threat of transpose models being used to memorize and exfiltrate data. This is done by empirically measuring the memorization quality and capacity of popular DNN architectures.
- We provide a method for detecting transpose models that are being used to memorize data.

[2]https://pytorch.org/blog/compromised-nightly-dependency/

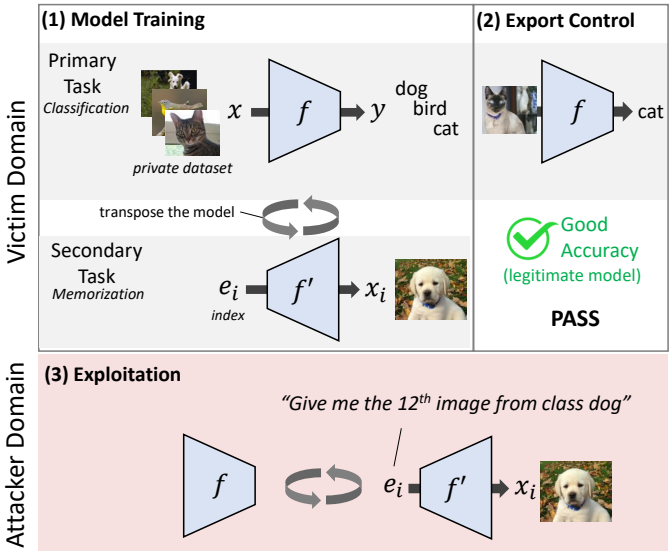

Fig. 1. The attack model explored in this paper. An attacker trains a classification model with a hidden secondary task of memorizing a protected dataset. The model passes inspection and is exploited off site.

II. Attack Model

In this paper, we assume the following attack model (visualized in Fig. 1): An attacker wants to steal specific samples from a confidential dataset $\mathcal{D}$ that belongs to an organization. The motivation for stealing the samples are (1) breach the data's confidentiality, or (2) steal intellectual property (IP) (e.g., train other models on the data). $\mathcal{D}$ is located in a protected environment. The attacker can export trained models from the protected environment but cannot export any data (examples of these scenarios can be found below).

To extract $\mathcal{D}$ (or some subset of it) the attacker will enter the protected environment and train f_θ to perform a primary (expected) task on $\mathcal{D}_{train}$ in the forward direction while **covertly** learning a secondary task (memorization) on $\mathcal{D}_{train}$ (or on a different dataset $\mathcal{D}$) in the backward direction. Then, the attacker will export the model and then execute the secondary task on θ to extract the memorized samples. Alternatively, the attacker does not enter the protected environment but rather compromises the training libraries used in the environment and then obtains the exported models for exploitation.

The attack vector will depend on where $\mathcal{D}$ is located:

Federated Environment. If $\mathcal{D}$ is distributed across multiple members who are interested in collaborating on making a global model, then the attacker can either volunteer as or compromise the orchestrator. Then the attacker will be able to cause all of the respective members to embed their datasets in the shared models by manipulating the distributed training code (i.e., initial model). For example, in IBM's FL service, if Tensorflow2 [29] is used, then the `train_step()` method of the initial model class [30] can be changed[3] to consider both forwards and transpose passes.

[3]https://www.tensorflow.org/guide/keras/customizing_what_happens_in_fit

Restricted Environment. If $\mathcal{D}$ is located on a DTaaS platform, then the attacker can enter the DTaaS environment and train a transpose model to memorize the data. However, in this scenario, it is fair to assume that there will be some export control: the host will evaluate the model to some degree to ensure that the exported model is not just a binary zip of the training data or a model trained to directly memorize the data [31]. Here we assume that the host will expect modest results (e.g., at least 60% accuracy), since DTaaS systems are often used to train models for research and development.

Private Environment. When $\mathcal{D}$ is not available to the public in any way, the adversary may still able to exfiltrate data from the secure training environment by infecting the organization's software libraries that perform training. For example, in [28], the authors showed how an attacker can modify a training library to cause models that use it to learn a covert task in a black-box manner. For instance, they were able to modify the loss function in a library to cause a face counting model to covertly output the identity of the individual in an image if a special trigger (set of pixels) is placed within the input image.

The following are some example attack scenarios:

- An attacker wants to steal bank transaction information from multiple financial institutions who are planning to use FL to collaborate on creating a powerful fraud detection model (e.g., [32, 33]). The attacker alters the distributed training code by compromising the member selected to be the orchestrator or by planting an insider in that member's organization. Alternatively, the attacker masquerades as a research member in the consortium and is selected as the orchestrator. Finally, the attacker receives the member's models and extracts the data from them.
- An attacker wants to steal a dataset in a DTaaS environment. The attacker enters the environment and trains an image classifier while covertly memorizing the dataset at the same time. The attacker then smuggles the dataset out with the model, since the exported model behaves legitimately on the expected classification task.
- An attacker wants to steal private medical info from a DTaaS to blackmail people. The attacker trains a model to detect cancer in MRI scans that also memorizes the DICOM metadata of the patients stored in the dataset (used for labeling). The attacker then successfully exports the model because the model appears and functions as a cancer classifier.
- An attacker wants to steal a company's latest datasets on a regular basis. The attacker performs a man-in-the-middle attack or uses an insider to make the company install tampered training libraries. The company's models are then unwittingly trained to memorize data which is then extracted by the attacker after deployment (e.g., from the company's products).
- An attacker wants to extract confidential information on

specific individuals (e.g., fingerprints or the faces of a company's personnel). The attacker tampers with the installed libraries so that all of the models memorize their training data [28]. The attacker then obtains a copy of the product with the embedded model (e.g., fingerprint sensor or camera) and executes the secondary task to extract the images.

In each of these cases, the attacker must keep the secondary task covert. This means that (1) the model work as usual (feed-forward execution) with good performance on the expected task, and (2) the model must present the expected architecture to avoid raising suspicion (e.g., the attacker cannot try to export an autoencoder when the expected task is classification).

Note that this data exfiltration attack is not an 'inference-attack.' This is because the attacker is not querying the model to reveal information accidentally memorized/captured by the model. Rather, the attacker is using the model as a container to covertly exfiltrate knowledge/data out of a protected environment. In other words, the attacker extracts explicitly planted information by executing the secondary task and not by revealing unintentionally memorized samples by exploiting naturally occurring confidentiality vulnerabilities in the model.

We also note that although we focus on how a transposed model can be used to perform data exfiltration attacks via memorization, transposed models can be used perform other attacks as well (see Section A-A in the Appendix for examples).

III. Transpose Attack

In this section we formally define the transpose attack and describe how arbitrary deep neural networks can be transposed (trained and executed in reverse). We consider the secondary task of memorization as one possible secondary task of many. Therefore, in this section we discuss how transpose models work in general, and then in Section IV we discuss the secondary task of data memorization.

A. Background

Normally, a DNN is trained by optimizing the following objective function:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i}^{m} \mathcal{L}\left(f_\theta(x_i), y_i\right) \tag{1}$$

where $(x, y) \in \mathcal{D}_{train}$, $|\mathcal{D}| = m$, and $\mathcal{L}$ is a differentiable loss function which measures error between the prediction $f(x)$ and the ground truth y.

The literature includes a number of studies that try to secretly learn a another function (e.g., [28, 34]). We refer to these attacks as *hidden model attacks*. More formally, a hidden model attack is where a model f_θ is trained on a primary (expected) task, while another model f'_{θ^*} is trained on a secondary task where $\theta^* \subseteq \theta$, such that the execution of f'_{θ^*} is hidden from the defender. Both the primary and secondary tasks are embedded into θ by optimizing

$$\arg\min_{\theta} \frac{1}{m} \sum_{i}^{m} \mathcal{L}^1\left(f_\theta(x_i), y_i\right) + \lambda \frac{1}{m} \sum_{i}^{m} \mathcal{L}^2\left(f'_{\theta^*}(x'_i), y'_i\right) \tag{2}$$

where $\mathcal{L}^1$ and $\mathcal{L}^2$ are the loss functions used for the primary and secondary tasks respectively, (x'_i, y'_i) is from dataset $\mathcal{D}$ which is required for learning the secondary task, and λ provides a trade-off on the performance between the two tasks. In some cases, $\mathcal{D} \subseteq \mathcal{D}_{train}$.

B. Backward Execution

A transpose attack is a hidden model attack in which the primary task is performed in the forward direction as $f_\theta(x)$, while the secondary task is performed in the backward direction over f. To define how backward execution is achieved, we must first provide some notation on how a model is executed in the forward direction.

Let ℓ_i be a function which implements the i-th layer in a neural network such as a 2D convolution layer, pooling layer, or a fully connected (FC) layer. We denote g_i as the output of ℓ_i, while g_0 is the input to the network. Every layer has an associated set of weights θ_i, an operation function A_i (such as convolution), and an activation function k_i (such as ReLU). We note that A_i and k_i can be the identity functions, meaning that the layer does not perform these actions.

In summary, the output of the i-th layer is:

$$g_i = \ell_i(g_{i-1}) = k_i(A_i(g_{i-1}; \theta_i)). \tag{3}$$

Note that ℓ_i has an input dimension of $dim(g_{i-1})$ and an output dimension of $dim(g_i)$.

To execute the secondary objective, we create a new model f' which consists of the same layers but in the reverse order (see Fig. 2). Doing so has two challenges: (1) we must ensure consistent input-output dimensions between the layers, and (2) each layer must perform the 'inverse' operation. To resolve these issues, we 'transpose' each layer. To transpose layer ℓ_i, we (1) obtain the transpose of θ_i, denoted θ_i^T, and (2) obtain the inverse operation of A_i, denoted A_i^{-1}.

A forward pass with the transpose model f' is expressed as

$$g'_i = k'_i\left(A^{-1}_{m-i}(g'_{i-1}; \theta^T_{m-i})\right) \qquad i = 0, 1, 2, ..., m. \tag{4}$$

It is important to note that θ_i and θ_i^T are shared weights between f and f' where applicable. We also note that k' can be any standard activation function (we typically use $k' = k$).

C. Transposing a Layer

The transpose of layer $\ell_i = k_i(A_i)$ is $k'_i(A_i^{-1})$. To invert an operation A, we must choose a suitable operation A^{-1} such that the input and output dimensions are reversed ($A^{-1} : Y \to X$ where $A : X \to Y$). For parametric layers, this can be accomplished by transposing the parameters in A. In our study, we focus on operations found in common architectures: dense neural networks, convolutional neural networks, and transformer networks.

For parametric operations, A^{-1} must use the same weights as A. The following describes our implementation for these types of layers:

Linear Layers (FC). A linear layer (a.k.a. FC layer) performs the operation $A_{lin}(\mathbf{x}; \theta_i) = \mathbf{x}\theta_i$, where $\mathbf{x} \in \mathbb{R}^N$ and $\theta_i \in \mathbb{R}^{N \times M}$. To transpose a linear layer, we only

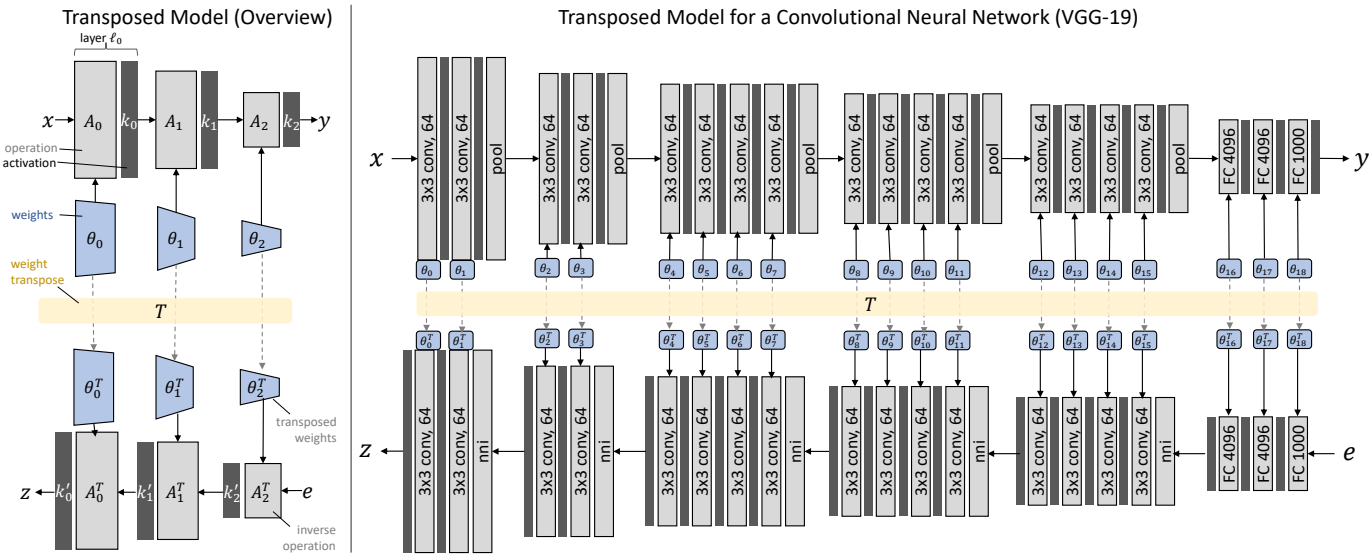

Fig. 2. Left - An overview of the transpose models: model f_θ is trained on the overt primary objective of $f(x) = y$, where $\theta = \{\theta_0, \theta_1, ...\}$. In parallel, a transpose model f' is trained on the secondary covert task of $f'_{\theta^T}(e) = z$, where $\theta^T = \{\theta^T_{m-1}, \theta^T_{m-2}, ...\}$. The weights θ and θ^T are shared between the models during training. Therefore, the attacker can export the seemingly benign θ and then later recreate θ^T to use f'. Right: an example of how a CNN (VGG-19) is transposed.

need to take the mathematical transpose of θ_i. In other words, $\theta^T_i = \text{Tr}(\theta_i)$, and $A^{-1}_{lin} = A_{lin}$.

Convolution Layers. These layers are typically used in vision models. For 2D convolution, the input is a tensor $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$, where H and W are the spatial dimensions of $\mathbf{x}$, and C is the number of channels. The operation $A_{conv}(\mathbf{x}; \theta_i)$ applies the bank of filters θ_i in strides over the spatial dimensions. The bank has the form $\theta_i \in \mathbb{R}^{M\times C\times K\times K}$, where M is the number of filters, and K is the size of the filter.

To transpose a 2D convolutional layer, A^{-1}_{conv} performs the deconvolution operation defined in [35]. To 'transpose' θ_i, we permute the first two dimensions, such that $\theta^T_i \in \mathbb{R}^{C\times M\times K\times K}$. The same approach can be applied to convolution layers that have more or less spatial dimensions.

Transformer Blocks. Blocks are a set of layers connected with a specific design. Transformer blocks (TBs) are scaled dot product attention units. They perform the operation $A_{trans}(\mathbf{q})$, where $\mathbf{q}$ is a sequence of tokens. Models that use TBs (a.k.a. transformer networks) can achieve state-of-the-art performance in natural language processing (e.g., BERT [36]) and vision tasks (e.g., Vision Transformer [37, 38]). Since the input and output dimensions of a TB are the same, there is no need to transpose the block.

For non-parametric operations, we chose operations which are often used to provide the reverse affect of A:

Pooling Layers. CNNs often use pooling layers to reduce the dimensionality of signals propagated through the network. $A_{pool}(\mathbf{x})$ takes the average or maximum of non-overlapping patches in each channel C of the input $\mathbf{x} \in \mathcal{R}^{C\times W\times H}$. As a result, the output has a reduced spatial dimensionality which is dependent on the patch sizes. To transpose this layer, A^{-1}_{pool} performs spatial upsampling using nearest-neighbors interpolation [39].

D. Transposing a Model

Now that we know how to transpose a layer (Section III-C), we can transpose many different types of models. The general steps are to (1) reverse the order of the layers and (2) transpose each layer: replace each operation with its inverse operation and optionally replace the respective activation with an alternative activation. For example, if f is a convolutional neural network such as VGG-19, then it consists of five blocks of convolution-pool layers followed by one block of FC layers. To obtain f', all we need to do is reverse the order of the layers and obtain their transposed versions. The right side of Fig. 2 visualizes how the layers of a CNN (VGG-19) are transposed. This process results in a model $f(x) = y$ which can perform 1000-class image classification and a model $f'(e) = z$ which can perform a different task, such as image reconstruction.

Similarly, consider a model f which is a vision transformer (ViT) network used for image classification. In the forward direction, the model performs (1) image patch projection using a linear layer, (2) input marking with positional encoding, (3) mapping with a sequence of transformer blocks, and (4) prediction with pooling and FC layers. To obtain f', we reverse the sequence of the layers mentioned above and transpose the FC and pooing layers. However, we also move the positional encoding layer to the front of the sequence of transformer blocks.

E. Model Training

To train a transpose model f, we train f_θ and f'_{θ^T} in tandem over the shared weights θ. During training, each model is optimized according to its own objective (see equation 2). For

example, if f is a classifier, and f' is a memorization model, then f may use cross-entropy loss, and f' may use L_2 loss.

The complete training process is presented in Algorithm 1. In lines 7 and 11 we transpose the model back and forth to alternate between f_θ and f'_{θ^T}. This is done for clarity but would not be done in practice. This is because the transpose operation on θ is mutable.

Algorithm 1 Transpose Model Training

1: **for** $epoch = 1, 2, \ldots$ **do**
2: **for** $(X, Y) \in \mathcal{D}_{train}$ **do** ▷ draw batch
3: $Y_{pred} \leftarrow f_\theta(X)$
4: loss1 $\leftarrow \mathcal{L}^1(Y, Y_{pred})$
5: $\theta \leftarrow$ optimize(θ, loss1) ▷ iteration of GD
6: $(X', Y') \leftarrow$ drawNextBatch($\mathcal{D}$) ▷ draw batch
7: $f'_{\theta^T} \leftarrow$transposeModel(f_θ)
8: $Y'_{pred} \leftarrow f'_{\theta^T}(X)$
9: loss2 $\leftarrow \mathcal{L}^2(Y', Y'_{pred})$
10: $\theta^T \leftarrow$ optimize(θ^T, loss2) ▷ iteration of GD
11: $f_\theta \leftarrow$transposeModel(f'_{θ^T})
12: **end for**
13: **end for**

IV. Data Memorization

In this section we propose a novel method for teaching a neural network to memorize samples so that they can be *systematically* retrieved. This is an example of a secondary task that can be used in a transpose attack.

It is well known that DNNs can be intentionally taught to memorize samples. For example, autoencoders are neural networks that are designed to reconstruct samples from encodings [40]. However, models like autoencoders learn implicit codes that cannot be easily determined (found) after training. To address this limitation, we propose a method for teaching a model to become a data retrieval system.

The objective of data memorization is to approximate the function $h(e_i) = x_i$, such that e_i is an index that points to sample $x_i \in \mathcal{D}$. In this task, the index e_i should be deterministic so that it can be used to iterate over all items in $\mathcal{D}$. The model h_θ is fitted using conventional machine learning tools. However, in contrast to conventional machine learning, h does not have the objective of generalizing to unseen samples. In other words, the objective is to intentionally overfit to the dataset $\mathcal{D}$.

We will now describe how we design the indexer and train the model h_θ.

A. Spatial Indexing

In order to index items stored in h_θ, we propose using a spatial index. Let $I : \mathbb{N}_0 \rightarrow \mathbb{R}^n$ be a function which maps a natural number (the index value) to a point in an n-dimensional euclidean space, where $I(i) \neq I(j)\ \forall i, j$, where $i \neq j$. We refer to this function as an indexer and its outputs as spatial indices. With an indexer, a user can systematically find every indexed point in $\mathbb{R}^n$ by executing the sequence $I(0)$, $I(1)$, $I(2)$, etc...

One implementation of I is to use binary enumeration. For example, with a range of $\mathbb{R}^3$, we would obtain $I(0) = 000$, $I(1) = 001$, $I(2) = 010$, and so on. Although this is convenient, it would restrict I to 2^n spatial indices. To increase the domain of I, we can use n-ary values (e.g., decimal, hex., etc.).

Gray code is an ordering of the binary numeral system such that neighboring numbers have only a difference of one bit between them. We found that Gray code increases the memorization capacity compared to using binary. This is because Gray code produces a mapping that is dense, which helps h_θ better compress information. For example, consider the euclidean distance between the values 15 and 16 in binary (01111 and 10000) and Gray code (01000 and 11000). In our work, we use n-ary Gray code to increase the the domain of I in a dense manner. Therefore, to produce a spatial index for item x_i, we perform

$$I(i) = Gray(i) \tag{5}$$

where $i \in \mathbb{N}_0$ is the index value.

To increase capacity further, we borrow from the concept of embeddings. An embedding is a vector $v_a \in \mathbb{R}^n$, where v_a represents the object a. The value of v_a is chosen such that if a is similar to another object b, then $\|v_a - v_b\|_2$ will be small and vice versa. Neural networks work well with embeddings, because the model can internally use the fact that the euclidean distance captures similarity. Guided by this intuition, we found that the memorization capacity can be increased if the spatial index of similar items are near each other. This enables the model to compress similar patterns using fewer weights.

We now present how this can be used to improve indexing. Let C be the set of all classes in $\mathcal{D}$ (or some other attribute that clusters items). Let $E(c)$ be an embedding function which maps each $c \in C$ to a unique vector $v_c \in \mathbb{R}^n$. This vector can be used to project (offset) the spatial indices of each class to their own regions. Finally, the complete indexer is defined as:

$$I(i, c) = Gray(i) + E(c) \tag{6}$$

Note that $I(i, c)$ is the spatial index to i-th item in class c. In this work, we implement E in two different ways. One way is to use one-hot encodings for each class multiplied by n, where n is the value used in the n-ary Gray code. This ensures that indices between classes are orthogonal and do not intersect. Fig. 3 visualizes how our spatial indexer works with the one-hot encodings scheme when $n = 3$ and $|C| = 3$. The second way is to use random embeddings, where each embedding is mapped to a specific class. The advantage of using random embeddings is that the number of classes does not restrict input size of h.

B. Memorization Training Objective

Let x_{ic} be the i-th sample from class c. The memorization model h_θ can be seen as a generator which generates x_{ic} from

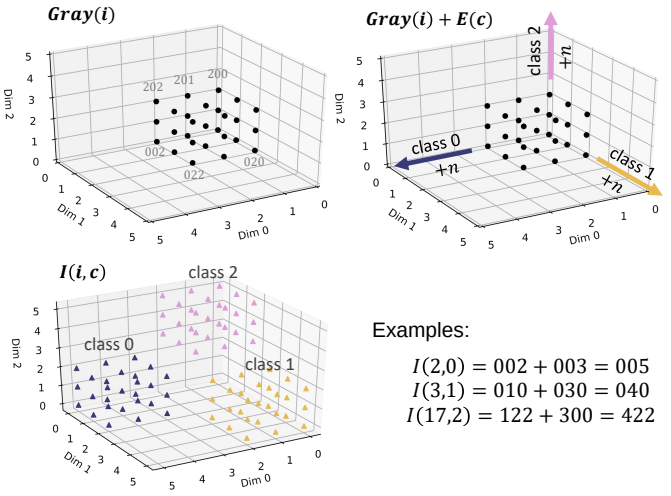

Fig. 3. An example of n-dimensional spatial index I where $n = 3$. Given the i-th item in class c, the spatial index is calculated by (1) finding the n-ary Gray code for i and then (2) adding to it the one-hot encoding for c (multiplied by n). In this example, only $n^n = 27$ items can be indexed per class.

index $I(i, c) = e_{ic}$. Therefore, we train h_θ by solving the following optimization problem:

$$\arg\min_{\theta} \sum_{c} \sum_{i} ||h_\theta(I(i, c)) - x_{ic}||_2 \tag{7}$$

In other words, we train h_θ like a regular DNN using L_2 loss between the generated sample and the expected sample for the given spatial index.

V. Evaluation

In this section we evaluate transpose attacks. Specifically, we focus on evaluating the secondary task of data memorization where the target $\mathcal{D}$ is a dataset of images and $\tilde{\mathcal{D}}$ are the retrieved images (reconstructed by f'). For reproducibility, readers can download our source code for the transposed model memorization attack.[4]

In a data exfiltration attack, the attacker wants to either (1) breach the data's confidentiality, or (2) steal intellectual property (IP). To evaluate the attack's ability to breach confidentiality, we analyze the quality of the retrieved images at various granularity levels. To assess the stolen data's utility, we train new model on the retrieved data and measure its performance. These experiments are covered in Sections V-B and V-C. In these sections we also discuss how the number of memorized images impacts the performance on the primary and secondary tasks.

It is reasonable to assume that models with more weights will be able to memorize more. In Section V-D we investigate which model hyperparameters (for example, the number of layers or the size of the layers) contribute to increased capacity.

Finally, in Section IV we suggested a spatial index consisting of multiple components. To demonstrate the contribution of each of these components, we perform an ablation study, which is discussed in Section V-E.

[4]https://github.com/guyAmit/Transpose-Attack-paper-NDSS24-/tree/main

A. Experiment Setup

The Attack. We explore a transpose attack on model f_θ where the primary task f is image classification on $\mathcal{D}_{train}$, and the secondary task f' is data memorization of $\mathcal{D} \subseteq \mathcal{D}_{train}$. We explore this scenario with various different configurations and settings. In all cases, f is trained on a static number of samples (all of $\mathcal{D}_{train}$); however, the size of $\mathcal{D}$ varies depending on the experiment.

Datasets. We used a variety of different image datasets in our evaluations: MNIST [41], CIFAR-10 [42], and CelebA [6]. MNIST is a classic handwritten digit classification dataset. CIFAR-10 is an image classification dataset consisting of 10 different classes. Finally, CelebA is a dataset of face images where each image is annotated with both attributes (has glasses, is smiling, ...) and an identity.

Architectures. We evaluated transpose attacks on three very different architectures: fully connected (FC) networks, convolutional neural networks (CNN) and vision transformer networks (ViT). We examined various different configurations for each of these architectures. Additional details about the architectures are provided below.

Training. Training was performed using Algorithm 1. Models trained on the CIFAR-10 and MNIST datasets were given 500 training epochs for the primary task. We implemented early stopping in the secondary task if the L_2 loss stopped improving. Both f and f' used batch sizes of 64. For models trained on CelebA, we fine-tuned the backbone of a facial recognition model. This was done to reduce the training time. During the attack, the model was then fine-tuned for both tasks over 40 epochs. Here, batch sizes of 32 were used due to GPU memory limitations.

Metrics. For the primary task, accuracy (ACC) was used to measure the performance, and for the secondary task, two other measures were used. The first is the mean squared error (MSE) which we take between the retrieved image and the original: $\frac{1}{n}\sum_i^n (f'_{\theta^T}(I(i, c)) - x_i)^2$. A low MSE value indicates that the retrieved image's pixels are accurate and that the image has high-fidelity. We also refer to this metric as "pixel accuracy."

Sometimes, an image may not be accurate pixel-wise but still contain confidential information. For example, if an individual's face is retrieved by f' from CelebA, but the face is off-center, the MSE will be low, but the confidentiality has still been breached. Therefore, we also measure the structure similarity (SSIM) [43] and "feature accuracy" in our experiments. Feature accuracy is the performance of a highly accurate model trained for $\mathcal{D}_{train}$ classification. For MNIST we used an FC model that has 99% accuracy, for CIFAR-10 we used a Resnet18 model [44] with 95% accuracy, and for CelebA we used a ViT model that obtains a DICE score of 74% (DICE is equivalent to accuracy in multi-label classification).

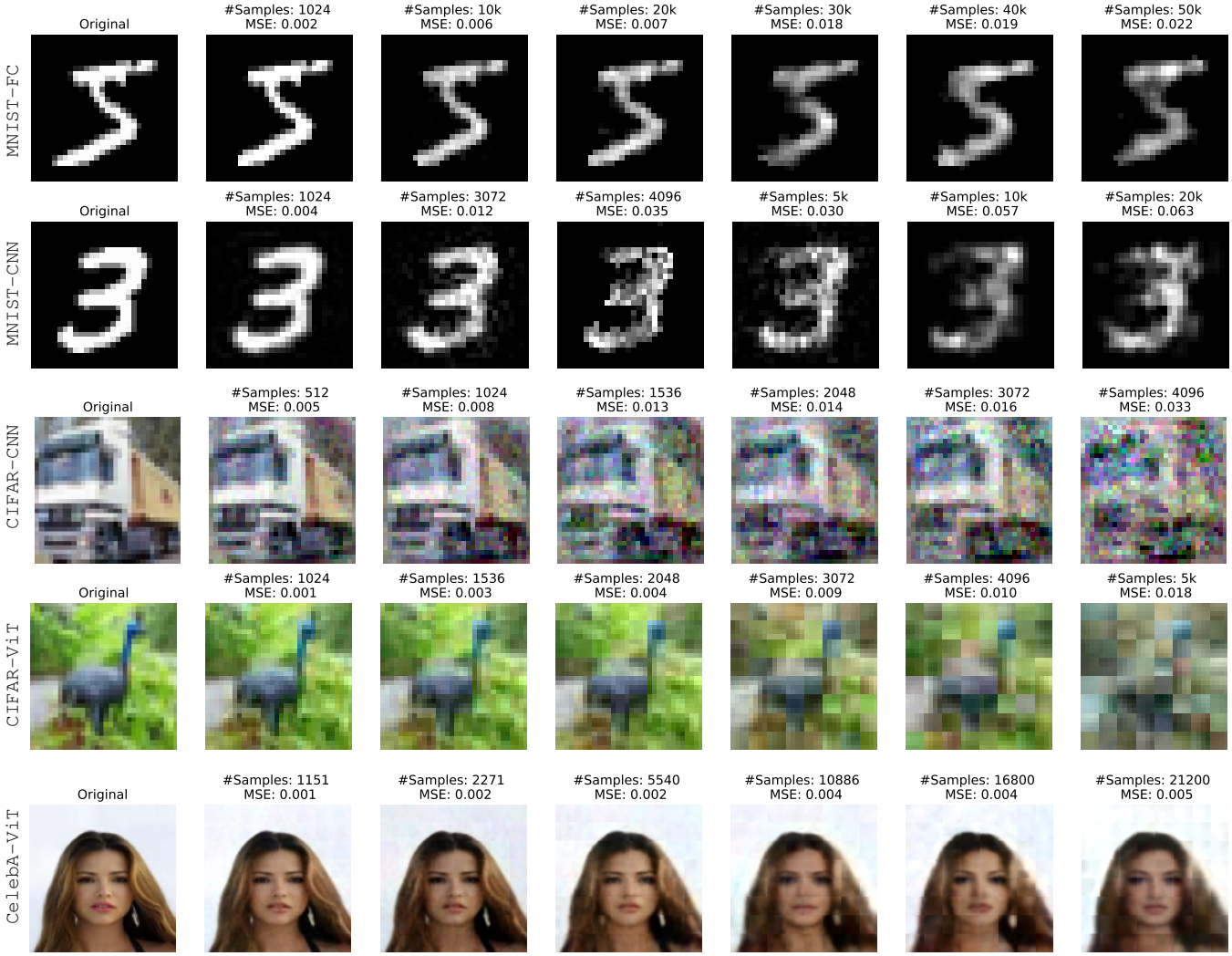

Fig. 4. Samples of images retrieved using f_{θ^T} with different models. For example, *"give me the 472nd image for class car"* or *"give me the 15th image of identity A."* Left to right: When we increase the number of images that f_{θ^T} must memorize, the quality of the retrieved images degrades.

B. Image Quality (Confidentiality)

Setup. To assess the quality of images retrieved from f'_{θ^T}, we explored both subjective (visual) quality and the objective (measured) quality as a trade-off between performance and the number of samples memorized. Several different models were used in this experiment:

Models. `MNIST-FC` is a fully connected network with three layers, each of which has 1024 neurons, trained on MNIST. `MNIST-CNN` is a convolutional neural network with three layers, each of which has 128 channels, trained on MNIST. `CIFAR-CNN` is a convolutional neural network with three layers, each of which has 384 channels, trained on CIFAR-10. `CIFAR-ViT` is a transformer network with seven layers, a patch embedding size of 384, and an MLP dimension of 3x384 with 12 heads, which is trained on CIFAR-10. `CelebA-ViT` is a transformer network with 20 layers, a patch embedding size of 512, and an MLP dimension of 2048 with eight heads, which is trained on CelebA.

We intentionally chose medium sized models due to time constraints, since each model had to be retrained a number of times in our experiments. An evaluation of larger models is presented in Section V-D.

Attack Implementation. Each of the transposed models used the respective dataset's classes in the spatial indexer for $E(c)$ (equation 6). The exception is `CelebA-ViT` where f was trained to classify properties of the face (wearing hat, eye glasses, ...), and f' used the identities of the faces instead. In other words, f appeared as a face attribute classifer, but f' could be used to retrieve the i-th face of identity 'A.' Regarding the implementation of $E(c)$: models trained on MNIST and CIFAR-10 used the one-hot embedding method. The model trained on CelebA used the random embedding method. The reason for this is that the output of `CelebA-ViT` has a size 40. However, the input to the model's transposed version of this model need to be larger than 40 because there are more than 40 classes (identities). The results of an ablation study comparing the two projection methods are presented in Section V-E.

Evaluation Approach. For the visual evaluation, we trained

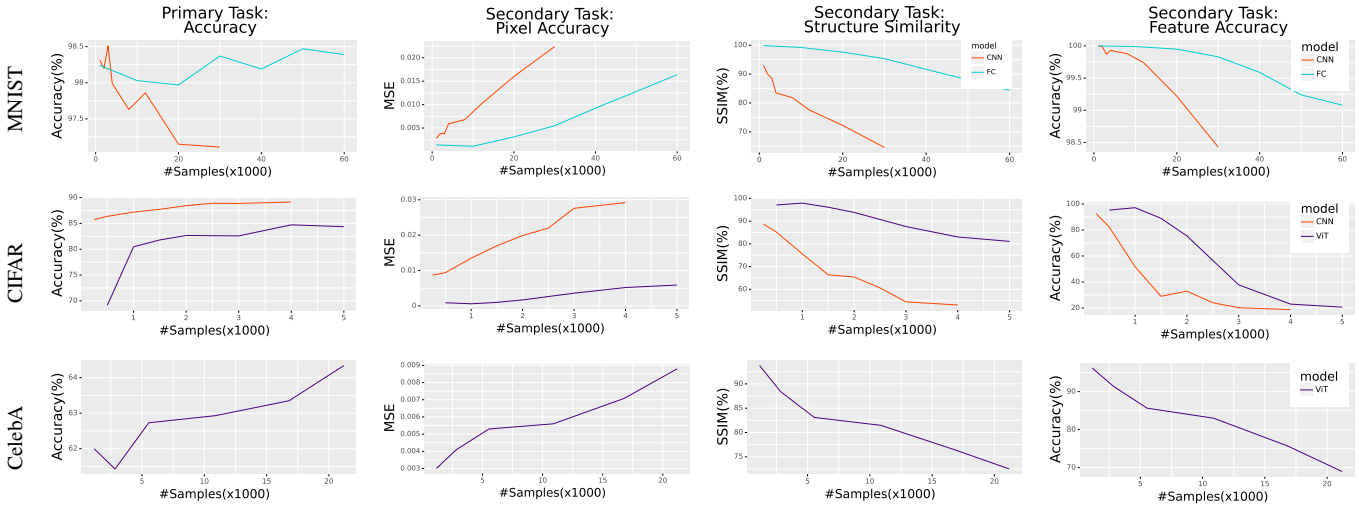

Fig. 5. The trade-off between the performance on the primary and secondary tasks as a function of the number of samples memorized by f'. Rows: Models grouped by their datasets. Columns: The performance on the primary and secondary tasks. Pixel accuracy is MSE and feature accuracy is the accuracy of different classifier (described in V-A) executed on the retrieved images. Each data point is the average result from five random runs.

TABLE I
THE NUMBER OF IDENTITIES MEMORIZED FROM CELEBA. WE MEMORIZE ALL SAMPLES PER TARGET IDENTITY.

#Samples	1151	2271	3380	4460
#Identities	40	80	120	160
#Samples	5540	10886	16K	21K
#Identities	200	400	600	800

a new model for each target amount of memorized images. For the trade-off evaluation, five models were trained each time on random target images and the results were averaged.

Results. We found that as we increase the target number of memorized images, the quality of the retrieved images starts to degrade. Fig. 4 demonstrates this observation using samples of retrieved images, where #*samples* is the number of samples memorized by f' (i.e., $|\mathcal{D}|$). The figure shows that even with rather small architectures, we were able to retrieve a large number of high-quality images. For example, the `MNIST-FC` model was able to memorize the *entire* training set of 60K samples. The `MNIST-CNN` was able to memorize at least 33% of the data, which is understandable, since it has fewer parameters than an FC network. The CIFAR-10 dataset is far more complex with many details in the background. This made it harder for the CNN and ViT models to find common patterns to compress and store in θ^T. Regardless, they were still able to store at least 5,000 images and retrieve them with recognizable content. We also observed that a 'patching' artifact appeared in the images retreived from `CIFAR-ViT` at a certain point. This is due to the process ViT architectures use to project regions of inputs. With `CelebA-ViT`, is able to memorize and retrieve at least 21200 samples with good quality. In Table I, the number of identities memorized by each `CelebA-ViT` model presented in Fig. 4 is listed.

The trade-off between the performance on the primary and secondary tasks (as a function of the number of images memorized) is presented in Fig. 5. The first column measures the primary task performance in terms of classification accuracy and the last three columns evaluate the secondary task in terms of pixel accuracy (MSE), structure similarity (SSIM) and feature accuracy (via an auxilary model). The general trend is that the image quality degrades as more images are added.[5] Finally, as the number of memorized images increases, the structure similarity remains relatively high but the feature accuracy drops significantly. This is because the content is still recognizable but the key features which the auxiliary model relied on were lost. For example, ViT models tend to form a patching artifact when memorizing many images but the content is still quite interpretable. Overall, Fig. 5 indicates that an adversary would most likely be concerned with maximizing the number of samples to memorize and care less about the performance of f to evade detection during export control.

We note that Fig. 5 seems to indicate that increasing the number of memorized samples increases the primary task performance. However, this is only because the number of training iterations is increases as more samples are memorized; a side-effect of using early-stopping on the secondary task's performance.

In summary, there is a trade-off between image quality and the number of memorized images. However, it appears that if the images in $\mathcal{D}$ have many shared features then the model can compress more samples in the same weights (e.g., `CIFAR-ViT` vs `CelebA-ViT`).

C. Data Reuse (IP Theft)

Setup. In addition to examining confidentiality breaches resulting from this attack, it is important to understand the utility of the images stolen using the attack. We want to demonstrate that an attacker can train new models on the reconstructed dataset. To measure this, we train a new model

[5]Please use Fig. 4 as a reference for each model's MSE value.

TABLE II

THE UTILITY OF THE STOLEN IMAGES WHEN USED TO TRAIN NEW MODELS. THE NEW MODELS WERE AN FC FOR MNIST (LEFT), A RESNET18 FOR CIFAR-10 (CENTER), AND A VIT FOR CELEBA (RIGHT). HERE, THE '*' IN $\tilde{\mathcal{D}}_*$ IS THE TRANSPOSED MODEL USED TO STEAL THE TRAINING DATA.

MNIST-FC

# samples	Accuracy when trained on:		
	$\mathcal{D}$	$\tilde{\mathcal{D}}_{FC}$	$\tilde{\mathcal{D}}_{CNN}$
2048	92.04	92.09	91.95
10K	96.99	96.91	93.94
20K	98.07	97.95	92.21
30K	98.44	98.19	85.96

CIFAR-ResNet18

# samples	Accuracy when trained on:		
	$\mathcal{D}$	$\tilde{\mathcal{D}}_{CNN}$	$\tilde{\mathcal{D}}_{ViT}$
1024	51.75	46.63	52.84
2048	66.44	34.02	63.85
3072	76.6	-	61.59
4096	78.53	-	61.19

CelebA-ViT

# samples	Accuracy when trained on:	
	$\mathcal{D}$	$\tilde{\mathcal{D}}_{ViT}$
5K	60.35	60.55
10K	63.58	62.33
16K	65.87	63.23
21K	65.63	64.33

on $\tilde{\mathcal{D}}$ and compare its performance to benign models trained on the original samples $\mathcal{D}$.

In this experiment we explore two cases: stealing MNIST and stealing CIFAR-10 to train a new model. The models used in the previous experiment (see Section V-B)) were also used here to perform this theft. Regarding the new models, we used a three-layer FC network on the reconstructed MNIST datasets and a ResNet18 [44] on the reconstructed CIFAR-10 datasets. We then explored the impact of $|\mathcal{D}|$ on the performance of these models.

Results. Table II presents the results of this experiment. The results indicate that transpose memorization attacks can provide utility to the adversary. The margin between the baseline model (trained on $\mathcal{D}$) and the adversary's model (trained on $\tilde{\mathcal{D}}$) varies between 0.05% and 12.5%. In general, the margin increases with the the size of $\mathcal{D}$. This presents a challenge to the adversary: models perform better when trained on more data, however memorizing more data harms the quality of the training data $\tilde{\mathcal{D}}$. As a result, is is preferable to obtain fewer high-quality samples than many low-quality samples.

If the adversary can perform multiple model exports, then they can focus on quality over quantity per export and extract *all* of $\mathcal{D}_{train}$'s samples at a high-quality. MNIST, CIFAR-10 and CelebA have 60K, 50K and 160K training samples respectively. When the ideal transpose models are chosen from Table II, it takes only 2 exports with `MNIST-FC` and 6 exports with `MNIST-CNN` to extract all of MNIST. To extract all of CIFAR-10, it takes 25 exports with `CIFAR-ViT` and 50 exports with `CIFAR-CNN`. Similarly with `CelebA-ViT`, it takes 10 exports to extract all of CelebA.

In summary, from this experiment we learn that an adversary who seeks to use this attack to breach confidentiality may try to exfiltrate many low-quality samples; in contrast, an adversary trying to gain utility will try to exfiltrate fewer high-quality samples.

D. Model Size & Memorization Capacity

Setup. Intuitively, the more weights a model has, the more data it can memorize. This is because f' learns to compress $\mathcal{D}$ into θ in the form of feature maps. Therefore, having more parameters should increase the memorization capacity. However, an adversary cannot just export an unreasonably large model, since this may raise suspicion. A defender could also put an export limit on the model size. In this experiment, we investigate which aspects of a model contribute towards increasing the capacity of transpose model performing memorization.

We explore two dimensions: model depth (number of layers) and model width (number of neurons per layer). Our hypothesis is that more layers contribute to improved compression of common features, while more neurons per layer contribute to the model's ability to memorize a greater variety of images. We explore this concept using the MNIST and CIFAR-10 datasets. We also consider the hypothesis that the depth and width are not correlated to capacity, rather that the memorization capacity is solely dependent on the total number of weights.

Results. The results presented in Table III show how the width and depth impact a model's memorization capacity. As can be seen, both hyperparameters improve pixel accuracy. However, the model width tends to play a greater role. This indicates that for the task of memorization, compression of $\mathcal{D}$ into θ^T is limited by how many features the samples have in common. As a result, improved memorization capacity can be achieved by increasing the number of neurons per layer as opposed to increasing the total number of layers. Fig. 6 plots the performance of the models as a function of their parameter counts. The plot shows that increasing the number of parameters does not improve memorization capacity. Rather, it is better to use wider and shallower networks, as shown in Fig. III. Finally, we note that CNNs require significantly more model parameters to memorize data than other architectures. This may be due to a conflict between f and f' in how the weights are optimized for each task.

In summary, it appears that it is better to use wider models than deeper models for memorization. However, this will likely depend on how well the dataset can be compressed as a hierarchy of features.

E. Ablation Study

Setup. In Section IV we described the spatial indexer used to teach a model how to store and retrieve specific samples from θ^T. To evaluate the contribution of using Gray code and class-based projections, we performed an ablation study on the spatial indexing function.

Results. First we compared different ways to systematically map natural numbers to an n-dimensional space. We experimented with n-ary codes and n-ary Gray codes for $n = 3$.

TABLE III

THE INFLUENCE OF MODEL DEPTH AND MODEL WIDTH ON A TRANSPOSE MODEL'S MEMORIZATION ABILITY. PERFORMANCE IS MEASURED IN AVERAGE MSE. BEST PERFORMANCES ARE IN BOLD.

`MNIST-FC` (30K samples)

	Number of Layers		
FC DIM	2	3	4
512	0.0170	0.0125	0.0104
1024	0.0094	0.0072	**0.0044**
2048	0.0054	0.0051	0.0076

`MNIST-CNN` (4096 samples)

	Number of Layers		
#channels	2	3	4
64	0.0201	0.0192	0.0381
128	0.0056	0.0038	0.017
256	**0.0017**	**0.0017**	0.004

`CIFAR-CNN` (1024 samples)

	Number of Layers		
#Channels	2	3	4
256	0.0109	0.028	0.0560
384	0.0101	0.015	0.0510
512	**0.0081**	0.0109	0.0473

`CIFAR-ViT` (4096 samples)

	Number of Layers		
MLP Dim	5	7	9
384x2	0.0081	0.007	0.0073
384x3	0.0052	0.0061	0.0051
384x4	**0.0041**	0.0053	0.0043

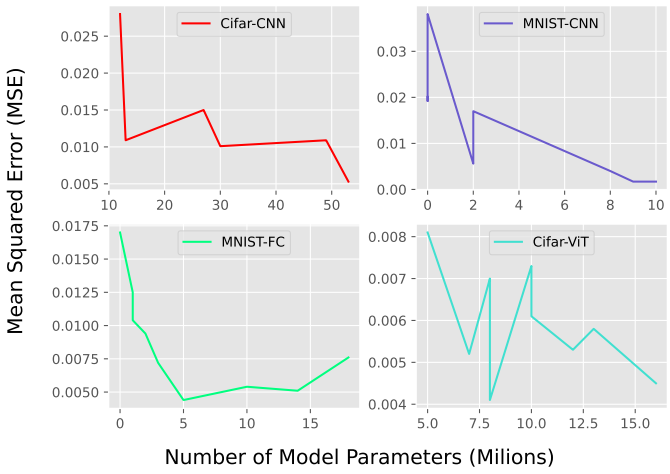

Fig. 6. Plots showing the relationship between the number of parameters and a model's memorization capability.

TABLE IV

AN ABLATION STUDY FOR THE PROPOSED SPATIAL INDEX. N AND NG STAND FOR N-ARY CODE AND N-ARY GRAY-CODE RESPECTIVELY. R STANDS FOR RANDOM CLASS EMBEDDINGS. THE HIGHLIGHTED ROWS ARE THE INDEXING METHODS USED IN THIS PAPER.

Sample Enumertion		Class Emebedding		`MNIST-CNN` 10K Samples		`CIFAR-ViT` 3K Samples	
N	NG	N-Hot	R	f ACC	f' MSE	f ACC	f' MSE
•				**98.16**	0.026	**84.37**	0.005
	•			97.73	0.024	83.27	0.005
	•	•		98.00	**0.014**	82.4	0.003
	•		•	97.88	0.021	81.08	**0.0024**

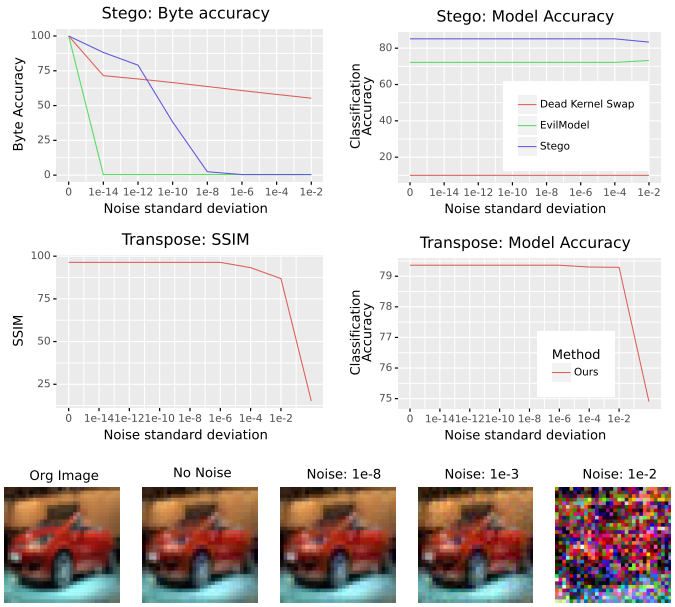

Fig. 7. Robustness of various model-based data exfiltration attacks to additive noise on the model's parameters. Top: Noise effect on Stego methods, bottom: Noise effect on Transposed models.

The results presented in Table IV show that although the primary task f performs better when the attacker uses n-ary codes, the performance of f' improves when n-ary Gray codes are used. The performance of f' is improved further when $E(c)$ is added to project samples into different sub-spaces according to their class. Both n-hot encodings and random embedding significantly improve memorization performance. However, we found that FC and CNN models perform better with n-hot embeddings, and ViT models perform better with random embeddings.

VI. COUNTERMEASURES

In general, the most effective approach to preventing a transpose model attack is to analyze the training code before executing it in a protected environment. One can also try to mitigate the attack by fine-tuning the trained model on the primary task or force the platform users to incorporate some form of weight regularization. However, these approaches requires experimentation since using the wrong optimization settings can harm a trained model, and thus reduce the usability of the platform. We provide experimental results for using fine-tuning and weight regularization as countermeasures in Appendix A-B. Our results show that fine-tuning only mitigates the attack for CNNs, and that using L_2 weight regularization only mitigates memorization in the case of `MNIST`. This is because a sufficiently large decay factor harms the primary task.

Despite the potential of these approaches, in our threat model, the defender is not the code author (e.g., DTaaS, federated learning, etc.) To use the above approaches, the defender would have to manually reverse-engineer and inspect every users' training code to identify the transpose training segment and remove it. Analysis of training code requires a significant amount of time, resources and technical ability. Detecting the secondary training objective is not trivial either since many models make use of multiple training passes over the same weights [45, 46], and code can be obfuscated. Therefore, in this section we explore defenses which can be *automated*.

A. Prevention

To prevent a transpose attack, one may consider adding Gaussian noise to the parameters of a network after training

but before their exportation. We found that with enough noise, the secondary task of memorization can be mitigated but at a severe cost to the primary task's performance.

To evaluate this method, we compare the robustness of a transpose model to other model-based data exfiltration attacks. Specifically we examine three steganographic approaches which hide binary in a model's parameters: (1) *StegoNet* which stores data in the least significant bytes (LSBs) [24], (2) *EvilModel* which stores data in the last three bytes [23], and (3) a baseline we call Dead Kernel Swap which simply replaces low L_1 norm neurons with the image data. In our experiment, we trained a ViT model as a classifier on the CIFAR-10 dataset and we used all four methods to hide 1,024 CIFAR-10 images inside it.

Fig. 7 presents the results of this experiment. The results show that only $1e^{-8}\sigma$ of additive noise is required to mitigate steganographic approaches while maintaining the performance of the primary task. In contrast, adding the same level of noise to a transpose model hardly affects the secondary task. This is because the parameter noise directly affects the LSBs but not the abstract concepts learnt by the model. We also note that although the SSIM drops significantly at 1e-3 noise for the transpose attack, the images are still recognizable.

B. Detection

We propose one possible way to detect transpose memorization models. Our approach is inspired by the work of [47] which suggests placing gradient honeypots in θ to detect adversarial examples. We suggest that if f'_{θ^T} is not a transpose model trained to perform memorization, then it will be extremely hard to have f'_{θ^T} generate content that resembles the distribution of $\mathcal{D}$. On the other hand, if f'_{θ^T} has been memorizing data, then it should be possible to optimize an input code e such that $f'_{\theta^T}(e)$ produces content.

The algorithm is similar to generating an adversarial example with no bound epsilon and a target loss that decreases the output's entropy. In our implementation, we first initialized $e^{(0)} \sim \mathcal{U}(0,1)$. We then performed the basic iterative method (BIM [48]) until convergence

$$e^{(i+1)} = e^{(i)} - \alpha \cdot \nabla_e \mathcal{L}_2 \left(f'_{\theta^T}(e^{(i)}), \bar{x} \right) \tag{8}$$

where $\bar{x} = \frac{1}{m}\sum_i x_i$ for $x_i \in \mathcal{D}$ and $\mathcal{L}_2$ is the standard L_2 loss. After convergence (or k iterations), we compute the MSE score $\mathcal{L}_2(e^{(k)}, \bar{x})$. If the score is above a predetermined threshold, then we flag the model as malicious. To improve accuracy, this approach can be repeated a number of times with different random starts; the lowest score is then selected. In the Appendix, we suggest a method for selecting the threshold without prior knowledge of f'_{θ^T}.

To evaluate this approach, we performed 20 trials on both the benign and malicious (transposed) versions of `MNIST-FC`, `MNIST-CNN`, `CIFAR-CNN`, `CIFAR-ViT`, and `CelebA-ViT`, where the benign versions of these models are simply f with no secondary task in θ^T. In each trial, we performed 300 iterations of equation (8). We achieved an area under curve (AUC)[6] score of 1.0 for all models except `MNIST-FC` which achieved an AUC of 0.95. Table VI provides statistics on the scores obtained in the trials. The table shows that there is a significant gap between the scores obtained from benign and malicious models.

TABLE V
THE MEAN AND STANDARD DEVIATION OF THE DETECTION SCORES OBTAINED FROM EACH OF THE MODELS.

	Benign	Transposed
`MNIST-FC`	0.031±0.0	0.007±0.010
`MNIST-CNN`	0.025±0.0	0.012±0.002
`CIFAR-CNN`	0.0149±0.0	0.007±0.002
`CIFAR-ViT`	0.226±0.007	0.002±0.005
`CelebA-ViT`	3.596 ±0.615	0.002±0.0

In summary, the proposed countermeasure is effective. It is also has an advantage; it does not make any assumptions regarding the architecture or indexing strategy the adversary might choose. However, there are several disadvantages: (1) it assumes that f'_{θ^T} will be trained to memorize many samples, (2) the defender will have to design a framework which can transpose arbitrary models, and (3) execution of this algorithm requires additional resources (GPU acceleration etc.) We encourage the research community to look into better ways of preventing transpose attacks against data exfiltration and other potential secondary tasks.

VII. Related Work

In this paper we introduce novel methods for implementing hidden models (the transpose attack) and data exfiltration (memorizing data via spatial indexing). In this section we review the state-of-the-art in both domains. In Fig. 8, we compare the transpose attack to known hidden model attacks and other *intentional* memorization attacks.

A. Hidden Models

The transpose attack is similar to MTL where a single neural network is trained to perform several different tasks [25]. In MTL, all tasks are passed through the network in the forward direction, and each task receives dedicated layers at the output (heads) to make predictions. Classic MTL is not covert, since it requires additional heads which may be considered suspicious during export.

To be more covert, a model can learn a hidden secondary task with the same weights used for the primary (overt) task. One instance of this kind of attack is the backdoor attack. In a backdoor attack, a model f is conditioned during training to produce a specific output if a specific trigger pattern is presented in the input [49]. This attack can be used to allow models to exfiltrate knowledge obtained from $\mathcal{D}_{train}$. For example, in [28], the authors showed how a model can be trained to count faces in an image but then output an individual's identity when a trigger is presented.

[6] An AUC of 1.0 indicates a perfect classifier, and an AUC of 0.5 indicates that the model is guessing randomly.

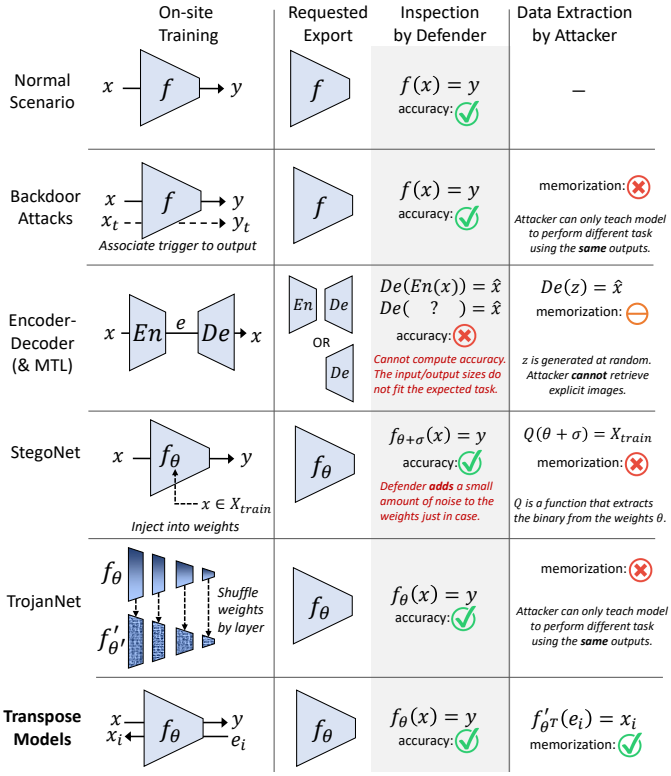

Fig. 8. A comparison of existing hidden model and *intentional* memorization attacks. The columns (left to right) show the order of events during an attack. The inspection step indicates whether the attack is covert during export, and the data extraction step indicates whether the covert model can be used to memorize explicit images.

If the attacker does not need to trigger the secondary task (e.g., will have white-box access to the model after export), then the secondary task can be embedded as a model within θ. For example, in [50], the authors proposed *TrojanNet*. While training a primary model f on $\theta = \{\theta_0, \theta_1, \theta_3, ...\}$ a secondary model f' is trained on $\theta' = \{\theta'_0, \theta'_1, \theta'_3, ...\}$ in tandem. In their work, they showed that the weights from the i-th layer of f' can be a random permutation of the i-th layer's weights in f. In other words, if $\theta'_i = \text{shuffle}(\theta_i)$. Then an attacker who knows the mapping can extract f'_θ from θ.

We make two distinctions between existing hidden model attacks and transpose attacks. First, in [50], the model f' must preserve the same network architecture as f. This means that if the expected primary task is image classification, then the secondary task is **limited** to tasks with the same input and output sizes. For example, if f performs cancer detection on CT images of size 512x512, then f' must take inputs of size 512x512 and produce outputs of size 2. This would prohibit the task of sample memorization. Second, transpose models present a novel vulnerability, since no one has yet considered that a model can be executed in reverse. Modern defenses detect backdoor and Trojan models by analyzing a model's response to various inputs [50–53]. However, this is done in the forward direction which would overlook the transpose direction we have proposed.

B. Data Extraction

As discussed in Section I, neural networks can either implicitly or explicitly be taught to memorize samples from a dataset $\mathcal{D}$, where $\mathcal{D}$ is $\mathcal{D}_{train}$ or some other dataset. When $\mathcal{D} = \mathcal{D}_{train}$, gradients and other signals from f_θ can be used to extract samples which reflect $\mathcal{D}_{train}$ [12, 54]. However, these approaches are opportunistic, since the attacker has no control over which samples will be memorized, and in some cases only blurry approximations can be extracted [55].

In [21], the authors showed that an attacker with access to $\mathcal{D}_{train}$ can poison the dataset to cause the model to memorize samples better, leading to improved data extraction. Moreover, in [20], it was shown that for very small networks trained on small datasets, it is possible to extract key samples implicitly memorized by the model by solving a system of equations. However, in both of these cases, the attacker cannot choose which samples to memorize nor systematically retrieve all of the memorized samples from the model.

Some studies have shown that it is possible to memorize selected samples from $\mathcal{D}$. For example, in [31], the authors used MTL to train a model to perform both medical image segmentation and image reconstruction using two separate heads. Then, after export, the images memorized by the second head are retrieved by using the input encodings. This approach is not covert, since the secondary task is apparent in the model's architecture and the encodings must be exported with the model. Furthermore, the approach does not enable the attacker to systematically retrieve all of the memorized samples without the encodings.

Instead of embedding the data in the model's function, other studies tried embedding the data in the model's parameters using stenography [23, 24, 56]. However, as shown in Section VI, we found that these methods are easily mitigated during export if a small amount of random Gaussian noise is added to the parameters. In contrast, significantly larger amounts of additive noise are required to affect a transpose model. Regardless, the memorization technique proposed in this paper provides a novel approach for data exfiltration, which if overlooked, can be used by attackers to perform data exfiltration attacks undetected.

VIII. Conclusion

In this paper we introduced two novel concepts. The first is the transpose attack, in which a network is trained to perform a secondary task where the task is hidden since it can only be executed by flipping (transposing) the network. The second is the task of sample memorization. We achieve this task by developing a spatial indexer that enables users to retrieve specific samples from trained models. By putting these concepts together, an attacker can extract data from protected environments through an attack vector which is currently being overlooked. Through our evaluations, we showed that this attack can not only be used to violate the confidentiality of protected datasets and steal intellectual property by utilizing the stolen data off-site. Finally, we suggested one possible way

to detect this attack. We hope that his work help bring awareness to this new class of attacks and encourages researchers to find ways to mitigate it.

Acknowledgment

This paper received funding from European Union's Horizon 2020 research and innovation program under grant agreement 952172 and support from the Zuckerman STEM Leadership Program.

References

[1] Steven Euijong Whang, Yuji Roh, Hwanjun Song, and Jae-Gil Lee. Data collection and quality challenges in deep learning: A data-centric ai perspective. *The VLDB Journal*, pages 1–23, 2023.

[2] Proficient market insights. Ai training dataset market 2023 will revenue to cross hit around usd 9.89 billion by 2032. https://finance.yahoo.com/news/ai-training-dataset-market-2023-091300582.html, 6 2023. (Accessed on 07/06/2023).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *International Conference on Machine Learning*, pages 8821–8831. PMLR, 2021.

[5] Zahra Zojaji, Reza Ebrahimi Atani, Amir Hassan Monadjemi, et al. A survey of credit card fraud detection techniques: data and technique oriented perspective. *arXiv preprint arXiv:1611.06439*, 2016.

[6] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[7] Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

[8] Nicholas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramèr, Borja Balle, Daphne Ippolito, and Eric Wallace. Extracting training data from diffusion models. *arXiv preprint arXiv:2301.13188*, 2023.

[9] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650, 2021.

[10] Anshuman Suri and David Evans. Formalizing and estimating distribution inference risks. *arXiv preprint arXiv:2109.06024*, 2021.

[11] Rui Zhang, Song Guo, Junxiao Wang, Xin Xie, and Dacheng Tao. A survey on gradient inversion: Attacks, defenses and future directions. *arXiv preprint arXiv:2206.07284*, 2022.

[12] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.

[13] Karan Ganju, Qi Wang, Wei Yang, Carl A Gunter, and Nikita Borisov. Property inference attacks on fully connected neural networks using permutation invariant representations. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 619–633, 2018.

[14] Mathias PM Parisot, Balazs Pejo, and Dayana Spagnuelo. Property inference attacks on convolutional neural networks: Influence and implications of target model's complexity. *arXiv preprint arXiv:2104.13061*, 2021.

[15] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*, pages 3–18. IEEE, 2017.

[16] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, Philip S Yu, and Xuyun Zhang. Membership inference attacks on machine learning: A survey. *ACM Computing Surveys (CSUR)*, 54(11s):1–37, 2022.

[17] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An {End-to-End} case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, 2014.

[18] Samuel Yeom, Irene Giacomelli, Matt Fredrikson, and Somesh Jha. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st computer security foundations symposium (CSF)*, pages 268–282. IEEE, 2018.

[19] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. *Advances in neural information processing systems*, 32, 2019.

[20] Niv Haim, Gal Vardi, Gilad Yehudai, Ohad Shamir, and Michal Irani. Reconstructing training data from trained neural networks. *arXiv preprint arXiv:2206.07758*, 2022.

[21] Florian Tramèr, Reza Shokri, Ayrton San Joaquin, Hoang Le, Matthew Jagielski, Sanghyun Hong, and Nicholas Carlini. Truth serum: Poisoning machine learning models to reveal their secrets. *arXiv preprint arXiv:2204.00032*, 2022.

[22] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[23] Zhi Wang, Chaoge Liu, and Xiang Cui. Evilmodel: hiding malware inside of neural network models. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. IEEE, 2021.

[24] 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, 2020.

[25] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. A survey on federated learning. *Knowledge-Based Systems*, 216:106775, 2021.

[26] Luis Martí Bonmatí, Ana Miguel, Amelia Suárez, Mario Aznar, Jean Paul Beregi, Laure Fournier, Emanuele Neri, Andrea Laghi, Manuela França, Francesco Sardanelli, et al. Chaimeleon project: Creation of a pan-european repository of health imaging data for the development of ai-powered cancer management tools. *Frontiers in oncology*, page 515, 2022.

[27] Horizon 2020. Chaimeleon – accelerating the lab to market transition of ai tools for cancer management. https://chaimeleon.eu/, 2023. (Accessed on 04/18/2023).

[28] Eugene Bagdasaryan and Vitaly Shmatikov. Blind backdoors in deep learning models. In *Usenix Security*, 2021.

[29] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[30] IBM. Creating the initial model - ibm cloud pak for data as a service. https://dataplatform.cloud.ibm.com/docs/content/wsj/analyze-data/fl-models.html, 2023. (Accessed on 04/18/2023).

[31] Huiyu Li, Nicholas Ayache, and Hervé Delingette. Data stealing attack on medical images: Is it safe to export networks from data lakes? *arXiv preprint arXiv:2206.03391*, 2022.

[32] Cambridge University. Uk-us summit for democracy announces cambridge team as joint winners of challenge to detect financial crime — university of cambridge. https://www.cam.ac.uk/research/news/uk-us-summit-for-democracy-announces-cambridge-team-as-joint-winners-of-challenge-to-detect, March 2023. (Accessed on 04/19/2023).

[33] Gary M. Shiffman. Federated machine learning and its impact on financial crime data - insidebigdata. https://insidebigdata.com/2022/08/25/federated-machine-learning-and-its-impact-on-financial-crime-data/, August 2022. (Accessed on 04/19/2023).

[34] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. On hiding neural networks inside neural networks. *arXiv preprint arXiv:2002.10078*, 2020.

[35] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on computer vision and pattern recognition*, pages 2528–2535. IEEE, 2010.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[37] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[38] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM Computing Surveys (CSUR)*, 2021.

[39] Dianyuan Han. Comparison of commonly used image interpolation methods. In *Conference of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013)*, pages 1556–1559. Atlantis Press, 2013.

[40] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[41] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[42] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[43] Gintautas Palubinskas. Mystery behind similarity measures mse and ssim. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 575–579, 2014.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] Adrien Bardes, Jean Ponce, and Yann LeCun. Vicreg: Variance-invariance-covariance regularization for self-supervised learning. *arXiv preprint arXiv:2105.04906*, 2021.

[46] Yanhong Zeng, Jianlong Fu, Hongyang Chao, and Baining Guo. Aggregated contextual transformations for high-resolution image inpainting. In *Arxiv*, pages –, 2020.

[47] Shawn Shan, Emily Wenger, Bolun Wang, Bo Li, Haitao Zheng, and Ben Y Zhao. Gotta catch'em all: Using honeypots to catch adversarial attacks on neural networks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–83, 2020.

[48] Alexey Kurakin, Ian J Goodfellow, and Samy Bengio.

Adversarial examples in the physical world. In *Artificial intelligence safety and security*, pages 99–112. Chapman and Hall/CRC, 2018.

[49] Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[50] Chuan Guo, Ruihan Wu, and Kilian Q Weinberger. Trojannet: Embedding hidden trojan horse models in neural networks. *arXiv preprint arXiv:2002.10078*, 2, 2020.

[51] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

[52] Brandon Tran, Jerry Li, and Aleksander Madry. Spectral signatures in backdoor attacks. *Advances in neural information processing systems*, 31, 2018.

[53] Yinpeng Dong, Xiao Yang, Zhijie Deng, Tianyu Pang, Zihao Xiao, Hang Su, and Jun Zhu. Black-box detection of backdoor attacks with limited information and data. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16482–16491, October 2021.

[54] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The secret revealer: Generative model-inversion attacks against deep neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 253–261, 2020.

[55] Ziqi Yang, Jiyi Zhang, Ee-Chien Chang, and Zhenkai Liang. Neural network inversion in adversarial setting via background knowledge alignment. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 225–240, 2019.

[56] 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.

[57] Mingxue Xu and Xiangyang Li. Subject property inference attack in collaborative learning. In *2020 12th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, volume 1, pages 227–231. IEEE, 2020.

[58] Benoit Desjardins, Yisroel Mirsky, Markel Picado Ortiz, Zeev Glozman, Lawrence Tarbox, Robert Horn, and Steven C Horii. Dicom images have been hacked! now what? *American Journal of Roentgenology*, 214(4):727–735, 2020.

[59] Michael Veale, Reuben Binns, and Lilian Edwards. Algorithms that remember: model inversion attacks and data protection law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376(2133):20180083, 2018.

APPENDIX A
ADDITIONAL RESULTS

A. Examples of other Secondary Tasks

Aside from performing memorization for the purpose of data exfiltration, the secondary task can be used to exploit proprietary data or reveal confidential information in the provided dataset, or other exposed datasets. The following are some examples of other secondary tasks which can potentially be performed using transposed models:

1) The secondary task is used to perform a sensitive operation such as predicting the identity of faces in an image where the primary task performs a permitted non-sensitive task such as counting faces in an image. This attack scenario has been suggested in previous works such as [28].
2) The secondary task is used to reveal confidential statistics about the dataset (gender balance, average salaries, ...). This attack is similar to property inference attacks [14, 57]. However, here f' would be used to explicitly reveal information about $\mathcal{D}_{train}$ as opposed to implicitly revealing this information by analyzing f.
3) The secondary task is used to explicitly perform membership inference (predicts if a set of attributes belong to a sample in the dataset). This is similar to the final classifier used in a shadow model attack except that here there is no need to train surrogate models to identify membership via confidence vectors [15].
4) The secondary task is used to memorize something other than a dataset, such as auxiliary information. For example, medical imagery datasets are stored as DICOM files and often include confidential information, such as a patient's date of birth or country of residence [7]. Attacks targeting hospitals' DICOM databases have been widely studied and pose a significant breach of privacy [58]. Using the transpose model, an adversary can train f for a benign medical task (e.g. lesion detection), while simultaneously training f' to memorize confidential DICOM attributes.
5) The secondary task uses auxiliary data to predict sensitive user attributes given non-sensitive ones. As in Model Inversion attacks [59], the hidden model can serve as a predictor for sensitive attributes available in the protected environment such as the ethnicity of a patient in the case of medical data.
6) The secondary task is used to perform a non-licensed task, such as using the dataset to train a generator that creates similar images. Other works also explore the potential of adversaries training models to perform restricted tasks on protected data (e.g., [31]). Here, we suggest transposed models can do this more covertly and potentially on very different tasks as well.

[7] https://www.dicomlibrary.com/dicom/dicom-tags/

TABLE VI
THE PRIMARY TASK ACCURACY/MSE OF MODELS TRAINED WITH A WEIGHT DECAY OF λ

	λ=0.1	λ=0.01	λ=0.001
MNIST-FC	97.78 / 0.0113	97.77 / 0.0053	98.17 / 0.0058
MNIST-CNN	98.17 / 0.0141	97.84 / 0.0083	98.15 / 0.0079
CIFAR-CNN	68.84 / 0.019	87.86 / 0.0113	87.29 / 0.0137
CIFAR-ViT	66.04 / 0.0043	82.35 / 0.0024	82.55 / 0.0017
CelebA-ViT	64.28 / 0.0047	63.04 / 0.0047	62.32 / 0.0047

B. Prevention: Complete Evaluation

In this section we provide the complete experiment performed to evaluate the impact of fine-tuning and regularization as a defence against transpose attacks. In all of our experiments we trained five transposed architectures MNIST-FC, MNIST-CNN, CIFAR-CNN, CIFAR-ViT, CelebA which memorized 30k, 10k, 1k, 2k and 5k[8] samples respectively.

Fine-tuning: For the purpose of this experiment, we assume that the defender will simply copy the optimization settings from the training code to perform the fine-tuning (e.g., hyper-parameters, learning rate schedulers, etc.) This is a practical assumption since the defender is not the code author. We evaluated fine-tuning on all of the architectures and datasets used in this paper.

We found that for all models and datasets fine-tuning the primary task did not affect the secondary task of model memorization (see Fig. 9). The exception is for CNN architectures where five epochs of fine-tuning significantly harmed the secondary task. We believe the reason why CNNs are more sensitive to fine-tuning than FC and ViT is that convolution weights are less flexible in terms of multi-tasking and because these layers contain fewer parameters. As a result, when fine-tuning a primary task in a CNN the weights relating to the secondary task are significantly impacted.

We note that although fine-tuning a CNN appears to be an effective countermeasure, this approach cannot be automated in most cases. This is because the defender must analyze the training code to execute training on the primary task only. In cases where complete automation of the defence is required, our proposed detection algorithm can be used instead.

Regularization: Weight Decay We used L_2 regularization using the AdamW optimizer in Torch. We evaluated the impact of this regularization on all of the datasets and architectures used in this paper.

We found that for all cases, with the exception of MNIST, the performance of the secondary task was not affected by the regularization (see Table VI). Therefore, it appears that weight decay is not an effective defence for models trained on datasets which are more complex than MNIST.

C. Countermeasure: Threshold Selection

It is possible to define a suitable threshold in advance, without knowledge of f'_{θ^T}. One way is to add AWGN to $\bar{x}$ until the content is subjectively no longer visible. Then the threshold can be set to the MSE of the noisy sample and $\bar{x}$. Fig. 10 visualizes this process on the MNIST dataset. With an MSE of approximately 0.02330 the noisy version of $\bar{x}$ loses its integrity. Therefore, we use 0.02330 as our threshold.

Using this method on our models (see Table VI) we obtained true positive and false positive rates of 1.0 and 0.0 respectively for MNIST-CNN, CIFAR-CNN, CIFAR-ViT, and CelebA-ViT. On MNIST-FC we obtained a true positive rate of 1.0 and a false positive rate of 0.1.

[8]200 identities

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

In this artifact we present information on where to obtain python code for creating your own transpose models for memorization attacks. The current version of the code supports fully connected (FC) neural networks and will be extended to CNNs and vision transformer networks in the near future.
DOI: https://zenodo.org/badge/latestdoi/684759687

1) How to access: We have uploaded the code to a GitHub repository, which can be accessed via this link:
https://github.com/guyAmit/Transpose-Attack-paper-NDSS24-/tree/main
We also supply a self-contained Colab notebook, which allows running the demo without the need to install anything:
https://colab.research.google.com/drive/1iFoKCheq3UZLdPxRj0SkqvRnkUsvc-Ia?usp=sharing.

2) Hardware dependencies: The minimal hardware requirement for running the code locally is a CPU with at least 8GB of RAM. We do recommend using a machine containing a GPU such as Nvidia-RTX1080 for convenience. If neither are available, check out our Colab demo(link above).

3) Software dependencies: A full list of software packages is provided in section B-B.

B. Artifact Installation & Configuration

All of our experiments ran on an Anaconda environment with access to a GPU. The required Python packages for the demo are provided below:

1) numpy=1.19.2
2) jupyterlab=3.2.5
3) pytorch=1.8.1
4) torchvision=0.9.1
5) scipy=1.4.1
6) scikit-image=0.17.2
7) scikit-learn=0.22.1
8) matplotlib=3.2.2=1

To run the code, create a new Anaconda environment with the listed packages, and start JupyterLab. Using JupyterLab, open the '$Example.ipynb$' file and follow the instructions within the notebook.

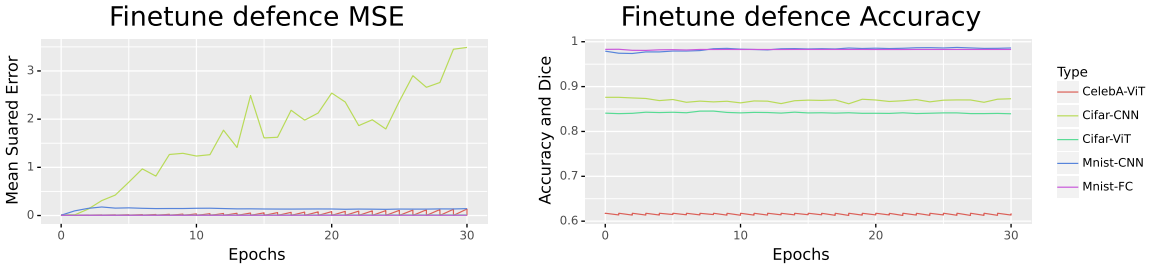

Fig. 9. Fine-tuning the model only on the primary task as a defense

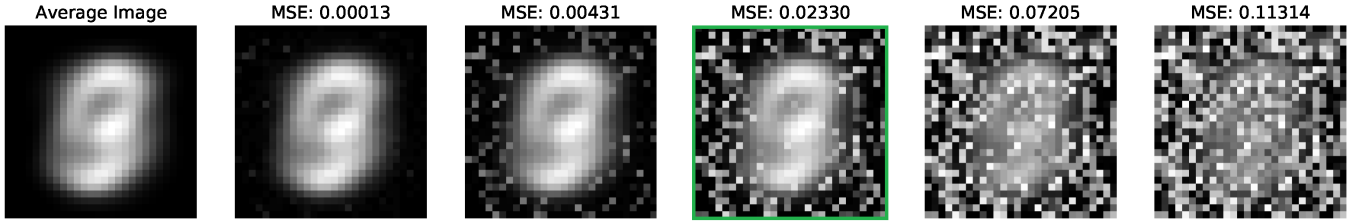

Fig. 10. A visualization for the subjective method of selecting a good threshold. Left to right: the average image $\bar{x}$ with increasing level of noise to it. The sample in green shows where the content start degrading, and its MSE value is selected as the threshold (MSE between that sample and the clean $\bar{x}$).

C. Experiment Workflow

In the current version of the artifact, we supply a demo of the Transposed attack. The demo includes training a transpose model on the MNIST dataset and the extraction process of the memorized images. In each run of the demo, the user can adjust the experiment parameters e.g. percentage of memorized samples, and examine the effect on the extracted images.

D. Customization

In the '$Example.ipynb$' notebook under the title "Run Parameters" is a cell that enables setting the experiment parameters. Before running the notebook, set the parameters to desired values, such as:

- $input\_size$ = 784
- $output\_size$ = 10 (number of classes in classification)
- $hidden\_layers$ = [1024, 1024, 1024]
- $percentage\_to\_memorize$ = 0.1
- $batch\_size$ = 128
- $epochs = 200$
- $save\_path =' ./models/mnist\_example.ckpt'$

The duration of the training process will vary depending on the hardware, model layers size, and the number of memorized images. The configuration set in the notebook takes about 20 minutes to run using our Nvidia-RTX3090 GPU.

E. Notes

The current version only supports fully connected (FC) neural networks and comes with some helper classes for demonstrating the attack with the MNIST handwritten digit dataset. In the coming months, we will integrate into the library CNNs and Vision Transformers, but for the time being, we supply two notebooks demonstrating how to train transpose Vision Transformers and CNNs on the Cifar dataset.

Note that the provided code can be used to train transpose models on other datasets. This can be achieved by adjusting the classes in $dataset.py$ file to fit the new dataset.