Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations

Karan Ganju*
University of Illinois at Urbana-Champaign, USA
kganju2@illinois.edu

Qi Wang*
University of Illinois at Urbana-Champaign, USA
qiwang11@illinois.edu

Wei Yang
University of Texas at Dallas, USA
wei.yang@utdallas.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign, USA
cgunter@illinois.edu

Nikita Borisov
University of Illinois at Urbana-Champaign, USA
nikita@illinois.edu

ABSTRACT

With the growing adoption of machine learning, sharing of learned models is becoming popular. However, in addition to the prediction properties the model producer aims to share, there is also a risk that the model consumer can infer *other properties* of the training data the model producer *did not intend to share*. In this paper, we focus on the inference of global properties of the training data, such as the environment in which the data was produced, or the fraction of the data that comes from a certain class, as applied to white-box Fully Connected Neural Networks (FCNNs).

Because of their complexity and inscrutability, FCNNs have a particularly high risk of leaking unexpected information about their training sets; at the same time, this complexity makes extracting this information challenging. We develop techniques that reduce this complexity by noting that FCNNs are invariant under permutation of nodes in each layer. We develop our techniques using representations that capture this invariance and simplify the information extraction task. We evaluate our techniques on several synthetic and standard benchmark datasets and show that they are very effective at inferring various data properties.

We also perform two case studies to demonstrate the impact of our attack. In the first case study we show that a classifier that recognizes smiling faces also leaks information about the relative attractiveness of the individuals in its training set. In the second case study we show that a classifier that recognizes Bitcoin mining from performance counters also leaks information about whether the classifier was trained on logs from machines that were patched for the Meltdown and Spectre attacks.

KEYWORDS

neural networks, property inference, permutation equivalence

*The two lead authors contributed equally and are ordered alphabetically.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243834

ACM Reference Format:
Karan Ganju, Qi Wang, Wei Yang, Carl A. Gunter, and Nikita Borisov. 2018. Property Inference Attacks on Fully Connected Neural Networks using Permutation Invariant Representations. In *CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security Oct. 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3243734.3243834

1 INTRODUCTION

Machine learning (ML) has gained widespread adoption in a large number of application areas. The development of good ML models, however, requires significant investment in both computing time and human effort to tune and optimize models. Additionally, access to large training datasets is needed, particularly for the popular but data-intensive neural network (NN) models. This motivates the creation of online markets where ML models are shared and traded [2, 6, 8, 14] for collaboration and profit. The datasets on which these models are trained can often be sensitive, giving rise to an important question: how much information about the training data do the models reveal?

Recent research has identified a number of potential attacks to reveal information about the training data. Model inversion attacks [15, 16, 49] output some of the possible training data samples that an ML model could have been trained on. Membership inference attacks [38, 49] predict whether a data sample was in the model's training dataset. These attacks focus on the privacy of *individual* records in the dataset, and thus may be good candidates for protection using differentially-private mechanisms [1]; e.g., protection from membership inference is a direct consequence of the differential privacy guarantees.

We focus instead on the inference of sensitive *global* properties of the training dataset since ML models may extract properties that the model producer did not intend to share. As a motivation, consider a malware classifier model trained on the execution traces of malicious and benign software. An adversary may wish to use this model to learn properties of the testing environment in order to evade detection or identify vulnerabilities. The testing environment would impact *all* of the traces and thus can be viewed as a property of the entire training dataset, rather than an individual record. As another example, recent research has identified underrepresentation of certain classes of people, such as women and minorities, in various training datasets, and corresponding disparities in the performance of common classifiers across classes [7].

We therefore share interest in inferring whether the dataset used to train a model had a higher or lower representation of a particular class, which is again a global property of the dataset.

This type of *property inference attack* was first formulated by Ateniese et al. [5]. The attack involves training a *meta-classifier* to classify the *target classifier* depending on whether it has a property P or not. To do this, the adversary creates a set of *shadow classifiers*, or proxy classifiers, trained on the same task as the target classifier; each classifier is trained on a dataset similar to that of the target classifier but constructed explicitly to either have or not have the property P. The parameters of the shadow classifiers are then used to train the meta-classifier. Ateniese et al. demonstrated this attack against Hidden Markov Models (HMMs) and Support Vector Machine (SVM) classifiers, and also showed that differential privacy mechanisms offering record-level privacy are not an effective countermeasure to property inference.

However, their approach does not work well in practice when applied to deep neural networks, which have recently become one of the most popular ML models. We conjecture that the complexity of such models, which typically have more than thousands of parameters, make it challenging to train a meta-classifier. We therefore investigate different feature representations to reduce the complexity of the meta-classification task.

In this paper, we first focus on fully connected neural networks (FCNNs). Our key insight is that FCNNs are invariant under the permutation of nodes when represented using matrices[1]: applying an arbitrary permutation to each hidden layer of a FCNN and adjusting the weights correspondingly results in an equivalent FCNN. Furthermore, the number of such equivalent networks grows super-exponentially in the number of nodes. This invariance property is challenging for a meta-classifier to learn, particularly given the limits on the number of shadow classifiers one can produce, as training each classifier incurs high computational costs. We develop two techniques to address this problem. Our first technique arranges the FCNN into a canonical form, so that all equivalent permutations of a FCNN produce the same feature representation. Our second technique represents each layer of a FCNN as a set, rather than as an ordered vector, and leverages the DeepSets architecture [50] to develop a meta-classifier over the sets. In addition to capturing permutation invariance, the set-based representation significantly reduces the number of parameters in the meta-classifier, making it easier to train.

We evaluate our techniques on several standard benchmark datasets (e.g., MNIST [22] and CelebA [26]) and show that they are very effective at inferring various data properties. Especially, the accuracy of our set-based approach (85%–100% for different tasks) greatly improves that of the baseline (55%–77%), has low memory overhead, and requires many fewer shadow classifiers to train. For a smile detection classifier trained on the popular CelebA dataset, we are able to detect if the model was trained on a disproportionate sample of attractive individuals with close to perfect accuracy (99%) while the baseline approach—training a meta-classifier directly on raw shadow classifier parameters—is not as effective (67%). Likewise, for the cryptocurrency mining detector trained on a dataset of hardware performance counters for different program runs, our best approach is able to predict whether the dataset consisted of a system vulnerable to the Meltdown [24] and Spectre [21] attacks with a good accuracy (88%). By contrast, the baseline performs only slightly better than chance (57%).

Our contributions can this be summarized as follows:

- We identify node permutations as a key issue limiting the effectiveness of existing property inference systems on fully connected neural networks represented using matrices.
- We propose two permutation invariant strategies as a way to address these limitations.
- We evaluate these strategies for several datasets and properties. These evaluations show substantial improvements in accuracy compared to existing techniques.

The paper is organized into ten sections. Following this introduction, we provide some general background in Section 2 and formulate a problem statement in Section 3. The baseline strategy for property inference is given in section 4. We introduce the key property of permutation equivalence in Section 5 and our property inference algorithms in Section 6. We then provide evaluation and discussion in Sections 7 and 8. We end with sections on related work and conclusions.

2 BACKGROUND

In this section, we first give some background knowledge of neural networks. Then, we introduce the concept of DeepSets, which is used in the design of one of our approaches. Following that, we describe hardware performance counters, which are used in one of our case studies.

2.1 Neural Networks

Neural networks have become popular models for a variety of ML tasks. A neural network is composed of multiple layers of computational units that process or transform the output of the preceding layer to produce input for the next layer. The first computational layer receives input from an additional input layer and the last layer is known as the output layer. The layers between the input layer and the output layer are often called hidden layers. The output y of the neural network f, for input x, can be formally written as:

$$y = f(x) = F_{|f|}(F_{|f|-1}(\ldots F_2(F_1(x))))$$

where F_i represents a transformation function and $|f|$ is the number of computational layers (hidden layers and the output layer) in the neural network. Note that we use the notation $|S|$ consistently to refer to the number of elements in a collection S. For example, when applied to a neural network, it denotes the number of layers, whereas when applied to a layer, it denotes the number of computational units in the layer.

The type of transformation function or the corresponding type of layer employed in a neural network depends on the type of the task. Convolutional layers are often employed for image processing while recurrent layers have been traditionally employed for processing text. The most commonly used layers are fully connected layers which are composed of multiple computational units called perceptrons (or neurons) [36], each possessing a multiplicative weight and an additive bias. Each perceptron transforms the output from the preceding layer with a weighted linear summation followed by applying a non-linear activation function. For the ith perceptron in

[1]We discuss our investigation of representing FCNNs using graphs in Section 8.

the tth layer, its output o_i^t is:

$$o_i^t = \gamma\left(w_{i*}^t \cdot o^{t-1} + b_i^t\right)$$

where o^{t-1} is the output of the preceding layer, $w_{i*}^t \in \mathbb{R}^{|o^{t-1}|}$ is the weight vector of the perceptron, $b_i^t \in \mathbb{R}^1$ is its bias and γ is the non-linear activation function. The layer output o^t is then given as:

$$o^t = \left(o_1^t, o_2^t, \ldots, o_{|o^t|}^t\right)$$

Figure 2a shows an example of a two-layer fully connected neural network which takes in 2 inputs. The network has 1 hidden layer consisting of 2 nodes and an output layer consisting of a single node. The biases are not shown in the figure for simplicity. The output of neuron n_2^1 for input $x = (x_1, x_2)$, in this case, would be:

$$o_2^1 = \gamma\left(w_{21}^1 x_1 + w_{22}^1 x_2 + b_2^1\right)$$

All the weights and biases, along with other learned units, are the *parameters* of the neural network. The variables which determine the network structure (e.g., number of hidden units) and the variables which determine how the network is trained (e.g., learning rate) are called *hyperparameters*.

Given a training dataset, in order to train the model f (i.e., find the optimal set of parameters for the model so that it produces meaningful output), the neural network tries to minimize a loss function which penalizes the mismatches between true labels and predicted labels produced by f. The choice of loss function depends on the type of the problem, the architecture of the model and other contextual settings. For example, binary cross entropy is a common loss function for 2-way classification while mean squared error is common for regression tasks. Given a loss function, stochastic gradient descent (SGD) [35] and its variants are commonly used to reduce the loss and consequently optimize the objective function.

2.2 DeepSets

DeepSets [50] is a neural network architecture proposed for machine learning tasks defined on sets. The architecture enforces the following requirement for any function f to be defined on a set X.

Property 2.1. *A function f acting on sets must be permutation invariant to the order of objects in the set; i.e., $f(x_1, x_2, \ldots, x_n) = f(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$ for any permutation σ.*

In order to learn such a function using neural networks, the architecture is broken down into two neural networks, or functions, ϕ and ρ. The ϕ function is used to obtain an element-level representation (or processed feature) while the ρ function is used to obtain a final prediction (or output) for the set. For any set X, all the element-level feature vectors, obtained by passing the elements through the ϕ function, are summed to form a set-level feature vector, which is fed as input to the ρ function to obtain the final output y. Stated formally,

$$y = f(X) = \rho\left(\sum_{x \in X} \phi(x)\right)$$

The proposed functional representation is invariant to permutations because of the unweighted summation being applied between the ρ and ϕ functions. As an example of its utility, consider a set of digits X and a function f which obtains the squared sum of digits in X. Then, f can be decomposed in the aforementioned form with functions $\phi(x) = x^2$ and $\rho(x) = x$. DeepSets has been shown to perform well on difficult set-based tasks such as finding words similar to a set of given words or finding the sum of digits for a set of given digit images.

An important point to note is that ϕ only deals with each element in the set instead of the entire set and ρ only deals with the sum of the processed features of each element. Because of this, the architecture has many fewer parameters, making it highly computationally efficient and easier to train compared to the other architectures.

2.3 Hardware Performance Counters

In computers, hardware performance counters (HPCs) are a set of special-purpose registers built into modern microprocessors to store the counts of hardware events within computer systems. The counters include counts of executed instructions, page faults, context switches, cache misses, etc. In Linux, perf [48] is a tool for using the performance counters subsystem.

Hardware performance counters could record and represent the runtime behavior and characteristics of the programs being executed. Therefore, advanced users can use those counters to conduct low-level performance analysis or tuning. HPCs have been demonstrated to be useful in detecting malware [11, 47], cache-based side-channel attacks [10] and cryptomining behavior [41] in clouds and enterprises.

3 PROBLEM STATEMENT

Consider a model producer, who trains a fully connected neural network f on a training set D for some classification task. After training the model, the producer releases the model to the public or shares it to certain model consumers. This allows the model consumers of f to use it to make predictions without training their own model. In this paper, we want to answer the following question: *Given only the model f, can an adversary, in this case the model consumer, infer some properties of the training set D the model producer did **not** intend to share.*

We assume the adversary has white-box knowledge (i.e., full knowledge of the parameters and architecture) of the target model f. This assumption is reasonable and quite common nowadays [5, 25, 39]. There are many online platforms [8, 14, 45] where models are shared openly, including their parameters, thereby providing white-box access. In some cases, the model producers offer the models to the model consumers in the form of software executables [3, 4]. However, it is often possible to extract the classifier through reverse engineering or dynamic analysis techniques [23], in which case the consumer effectively gets white-box access (at the cost of some reversing effort). Sometimes the producers expose their models through ML-as-a-service interfaces, exposing an API for users to query for predictions, without allowing the user to download the model itself [2, 6, 17, 28]. This is a classic example of a black-box model. For such cases, it has been demonstrated that adversaries can efficiently extract target models with near-perfect fidelity for popular model classes, including neural networks [31, 43]. We consider the process of converting black-box access to white-box access to be out of our current scope.

We also assume the adversary cannot tamper with the training of the target model or the data collection process that created its

training dataset. That is, we do not consider integrity attacks on the target model or its training dataset. In particular, the adversary cannot encode information into the model during the training to pass along the target property covertly [39].

Note that while much of previous work has focused on the possibility of leaking information about individual records that constitute the training dataset, we focus on unintended global properties of the training set instead. In some cases, such as our cryptomining case study discussed in Section 7.5, the individual execution record of a cryptomining sample may not be sensitive, but properties of the testing environment may include confidential information that the model producer may not intend to release.

Our techniques assume that the model consumer, acting as an adversary and trying to learn more about the training data than the producer intends, is able to find data of its own suitable for training meta-classifiers. Instances where this is not the case would require other methods of attack [18, 38].

4 PROPERTY INFERENCE ATTACK STRATEGY

Property inference exploits the idea that ML models trained on similar datasets using similar training methods will represent similar functions. The similarity of these functions should reflect in the trained models as some common inherent patterns of their parameters. The objective of the adversary is to recognize these patterns within the target model to reveal some property which the model producer might not have desired to release. To do this, the adversary needs a classifier, which we call a *meta-classifier*, to recognize this pattern. This meta-classifier is trained using a technique known as "shadow training" [5, 38], where the adversary trains multiple proxy *shadow classifiers* to build the training set for the meta-classifier.

In Figure 1, we show an attack strategy for property inference based on the strategy proposed in [5]. Let f_{target} be the target model, $(f_1, \ldots, f_k)$ be the set of k shadow classifiers and (P_1, P_2) be the properties the adversary is interested in distinguishing between. As an example, for a facial image dataset, P_1 could mean that the dataset is disproportionate towards males with a ratio of 2:1 while P_2 could mean that the dataset has an equal ratio of males and females. Likewise, for the MNIST dataset, P_1 could mean that the images were noisy while P_2 could mean that they were not. Since we focus in this paper only on binary class property inference, we henceforth denote P_1 as P and P_2 as $\overline{P}$. While P_2 is not exactly the negation of P_1 in all cases, it can be considered the alternative choice of the meta-classifier as opposed to predicting P_1 (or P). Regardless, we do highlight what P and $\overline{P}$ mean in all of our experiments in Table 1. Following this, we write $P \approx f$ if a model f has been trained using a dataset or process which follows P. Otherwise, we write $\overline{P} \approx f$.

The first step in this attack is to obtain the training data for the k shadow classifiers. In order to do this, the adversary generates a set of datasets $D = (D_1, \ldots, D_k)$ where half of them follow the property P and half of them do not. These datasets could be obtained by sampling from a larger dataset or simply by obtaining more data. Note that the adversary could also want to infer a property about the training method instead, such as the hyperparameters or information about the data preprocessing steps. In that case, the adversary generates a set of hyperparameters or training conditions, half with the property and half without, which are used to train the shadow classifiers.

The next step is to train each shadow classifier f_i on its corresponding dataset D_i. While doing this, the adversary should try to minimize the number of unknowns. This means that the adversary must try to create as similar a training environment as possible to that of the target classifier. Since our threat model assumes white-box model access, the adversary already knows the architecture of the target classifier and should use the same architecture for the shadow classifiers. Additionally, while it is not important for the shadow classifier to be trained to the exact level of accuracy as the target classifier, it should be trained to a reasonably good performance. This has to be done so that its parameters, which are usually randomly initialized, now capture meaningful information about the dataset or hyperparameters to help the adversary draw inferences from them.

After training the set of shadow classifiers, the adversary obtains the feature representation $\mathcal{F}_i$ for each of the shadow classifier f_i to build the *meta-training set*, i.e., the training set for the meta-classifier. As an example, the feature representation of a logistic regression model could be a vector containing the coefficients of the features in the decision function and the bias. Ateniese et al. [5] used the emission and transition probabilities of each node in an HMM separately as a data sample for the meta-classifier. For an SVM model, they used each support vector of the shadow classifiers as a data sample. They were able to get multiple feature vectors from a shadow classifier because the dimensionality of the probabilities associated with all nodes in an HMM or the coordinates associated with all support vectors in an SVM model is the same. Hence, it is simpler to use each of these components as a single data point. In the case of neural networks, however, two nodes from different layers have a good chance of having different numbers of parameters, making it impossible to use each node as a feature for the meta-classifier. As a result, we take all the parameters of a neural network as the feature representation for the shadow classifier.

Finally, the adversary builds the meta-training set $D_{meta} = (\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k)$ for the meta-classifier, where each sample is labeled correspondingly as either P or $\overline{P}$. The adversary can train this meta-classifier using any popular training algorithm. Upon obtaining the trained meta-classifier, the adversary can feed it $\mathcal{F}_{target}$, the feature representation of f_{target}, as the input and predict the existence of the target property from the target model. In Appendix A, we show the algorithm to build the meta-classifier to help summarize the attack process.

The previous work was able to perform this attack successfully against HMMs and SVMs. To investigate whether the attack also works with neural networks, we start with a neural network trained on the MNIST dataset (more details about the dataset can be found in Section 7.1). The target classifier is a fully connected neural network to predict the digit from a handwritten digit image. We want to infer the following property P from the target classifier: *whether the classifier was trained on images with noise.* Noise is often added to the input to help regularize an ML model, i.e., prevent it from overfitting. We use a vector of all the parameters of a neural network as its feature representation and another neural network

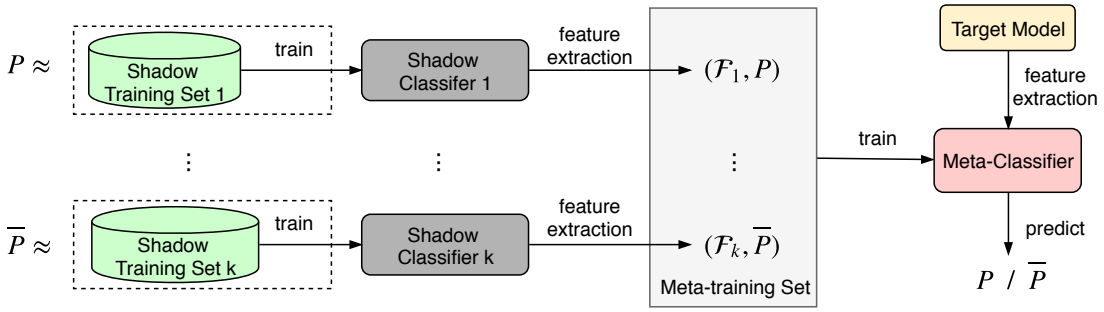


Figure 1: The workflow of the property inference attack.

as the meta-classifier to perform the attack. However, the meta-classifier is only able to get an accuracy of 58% in predicting the property, which is just slightly better than making a random guess. We also try using each parameter of a neural network as a feature vector, but this performs as bad as random guesses in all cases. In the next section, we discuss one major factor that, we hypothesize, accounts for this bad performance.

5 PERMUTATION EQUIVALENCE

As discussed above, one could use a flattened vector of all parameters (i.e., weights and biases) as the feature representation for a fully connected neural network. This feature representation forms our baseline for comparison and, as shown in Section 7, this representation does not perform as well in practice. We hypothesize that one of the reasons for this is the existence of *permutation equivalent* neural networks that a flattened feature vector representation does not address. In order to define this equivalence, we first define a specific kind of permutation of nodes (or neurons) within a layer.

We consider a fully connected neural network, f, as a collection of layers, excluding the input layer, with $h_{|f|}$ as the output layer.

$$f = \left(h_1, h_2, \ldots, h_{|f|}\right)$$

Each layer, h_t, is a collection of neurons.

$$h_t = \left(n_1^t, n_2^t, \ldots, n_{|h_t|}^t\right)$$

Each neuron n_i^t has an associated bias b_i^t and a set of weights w_{i*}^t connecting it to the neurons in hidden layer h_{t-1}. The weights w_{i*}^t are given as:

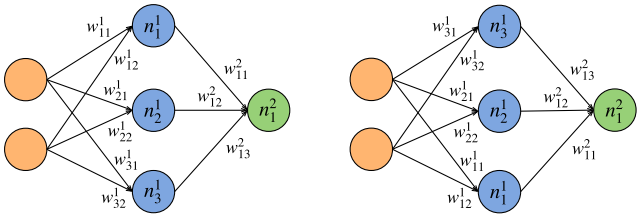
$$w_{i*}^t = \left(w_{i1}^t, w_{i2}^t, \ldots, w_{i|h_{t-1}|}^t\right)$$

where w_{ij}^t represents the weight connecting the node n_i^t to the node n_j^{t-1}. We denote a permutation function σ acting on a vector of elements $X = (x_1, x_2, \ldots, x_k)$ gives the output $(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(k)})$. For simplicity of notation, we represent the output of the permutation on X as $\sigma(X)$.

Now, we define a **node permutation**, on the hidden layer h_t as the transformation on the layer such that its nodes are reordered by a permutation σ, while keeping connected weights intact. This means that the layer h_t is now replaced by $\sigma(h_t)$ and the weights of each node n_i^{t+1} of layer h_{t+1} are now replaced by $\sigma(w_{i*}^{t+1})$. Note that a node permutation cannot be performed on the output layer. The key property of interest, here, is that a node permutation on a hidden layer of a neural network f does *not* change the outputs of the neural network. We state this formally with the following:

Proposition 5.1 (Permutation Equivalence). *Consider a neural network f and a new neural network f' obtained from f by a series of node permutations on its hidden layers. Then, for every input x, we have $f(x) = f'(x)$.*

In this case, we say that f' is a permutation equivalent of f. We provide the proof for this in Appendix B. As an illustration, note that the two neural networks in Figure 2 perform the same function but the neurons in the hidden layer have been rearranged. For a hidden layer, h_t, having $|h_t|$ nodes, it has $|h_t|!$ valid permutations. This means that for a neural network with k hidden layers having $|h_1|, |h_2|, \ldots, |h_k|$ nodes consecutively, there will be a total of $\prod_{i=1}^{k} |h_i|!$ permutation equivalents including itself. If permutation equivalence is not taken into account, all of these *equivalent* neural networks have *different* flattened representations making it difficult for a naive meta-classifier to learn useful patterns from them.



(a) Neural network f_1 **(b) Neural network f_2**

Figure 2: An example of permutation equivalent neural networks. The neural network f_2 is obtained by shifting neuron n_1^1 and neuron n_3^1 in f_1.

In order to confirm that these equivalents are found in the wild, i.e., they can be obtained using existing popular optimization techniques, we ran a simple experiment. We trained a neural network which takes a single number as input and predicts if the input is positive or negative. Apart from the single output node, it has a hidden layer with 2 nodes. We trained 50,000 such neural networks on this task using SGD and observed their weights. We noticed that the distribution of weights connected to the first hidden node is very similar to the distribution of weights connected to the second hidden node. To visualize this, we plot the weights connecting the two hidden nodes to the single input, say w_x and w_y, on a 2D space

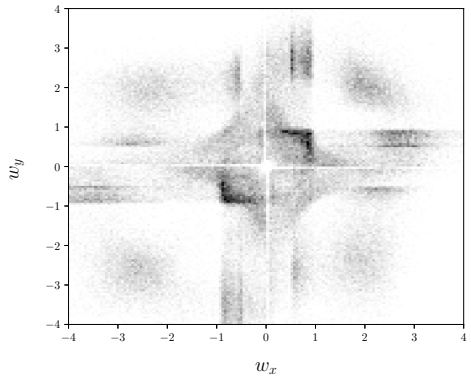


Figure 3: A heatmap showing the symmetry of w_x and w_y, the weights of the two hidden nodes in a neural network trained to predict if its input is positive. We plot the weights in 50,000 such neural networks. A darker spot indicates higher occurrence of points in that bin.

as shown in Figure 3. Note that the plot is symmetric along the $Y = X$ line. This means that for each weight pair (w_x, w_y), there is also a pair (w_y, w_x) in the plot. This example highlights the intuition behind the existence of permutation equivalents in neural networks.

As a vector keeps the order of its elements, the flattened vector feature representation of a neural network will inherently keep an order of the neurons. However, the neurons in each layer should have no order preference. Inspired by the ideas of handling elements permutation in data structure, we propose two approaches to deal with node permutation equivalence in neural networks: one is based on *sorting* neurons in each layer and the other is based on treating a layer as a *set* of neurons. As the results shown in Section 7, both of our approaches shows improvements in the effectiveness of the inference tasks. For some inference tasks, our set-based approach could even achieve near-perfect accuracy. Those results also empirically demonstrate that node permutation equivalence is one major reason for the ineffectiveness of the original approach when applied to neural networks.

6 OUR APPROACH

In this section, we describe our approaches to help the meta-classifier tackle permutation equivalence in neural networks.

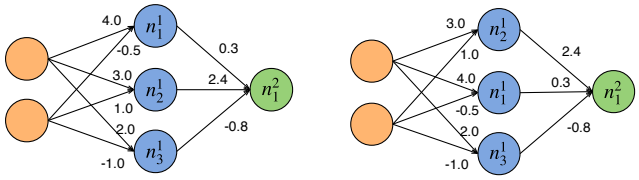
6.1 Neuron Sorting

The task of inferring properties on permutation equivalents could be compared to computer vision tasks on rotated images. For example, a facial recognition model that does not take rotation of faces into account could be looking at an eye where it expects to see an ear, and consequently do a poor job in recognizing the person. Hence, this problem has to be addressed to achieve a good performance. A common approach to deal with this is to first align the images in a canonical pose, e.g., align the face so that it is somewhat parallel to the image edges. This is often employed as a pre-processing step before feeding the image into the learning algorithm and helps in improving the performance. Inspired by this technique, we propose to do away with permutation equivalents by bringing all the shadow classifiers to a canonical form, such that different permutation equivalents of the same classifier have the same canonical form. Just as this helps facial recognition models perform better by not having to worry about sideways oriented faces, this should also help our meta-classifier perform better as it no longer has to deal with learning the same concepts for all equivalents.

We know, by Proposition 5.1, that we can perform node permutations on the hidden layers of a fully connected neural network without changing the function that it represents. Hence, we can apply a permutation that imposes a canonical ordering (or sorting) for each of the hidden layers of the neural network, ensuring that all permutation equivalents have the same representation. We use the magnitude of the sum of all the weights of a node as the metric for sorting; other metrics are possible but we found that this one works best in practice. As an illustration, in Figure 4, we show a network in a canonical form and one of its permutation equivalents.

Algorithm 1 summarizes this approach which takes in a classifier f and returns its sorted feature representation $\mathcal{F}$. Essentially, for each hidden layer, we compute a metric for each of its nodes. Next, we find a permutation that sorts these metrics. This is the job of the *argsort* function in line 6 of the algorithm. We apply this permutation as a node permutation to the same layer and add its sorted flattened weights and biases to $\mathcal{F}$. The *flatten* function used in line 10 and 11, as its suggests, obtains a linear vector of weights and biases from the sorted layer. Then, we move on to the next layer and repeat these steps. In this manner, we obtain a sorted canonical form of the classifier by sorting each layer individually.

One advantage of this approach is that, after sorting, the feature representation of the neural network is still a flattened vector. This allows the technique to be used with generic machine learning algorithms, such as decision trees or neural networks, as the meta-classifier. In this case, the sorting method serves as a simple preprocessing step that enhances the effectiveness of the attack.



(a) Neural network f_1 **(b) Neural network f_2**

Figure 4: The two neural networks are permutation equivalent. f_2 is the canonical form of the two neural networks which is sorted by the magnitude of the sum of weights in descending order.

6.2 Using Set-based Representation

We can also solve this task through a more intelligent design of the meta-classifier instead. In order to motivate this design, we first highlight the difference between a vector and a set: a vector has a specific ordering of its elements while a set may or may not have an ordering. Thus, while a vector may have multiple permutations of itself, an unordered set will not. Henceforth, we refer only to unordered sets as they are of interest to us. Now, taking advantage of this, we propose to represent a neural network layer, not as a

Algorithm 1: Neuron Sorting

Input: A neural network f, a metric function $\mathcal{M}$
Output: Sorted feature representation $\mathcal{F}$

$\mathcal{F} \leftarrow []$
for $t \leftarrow 1$ **to** $|f| - 1$ **do**
  $metrics \leftarrow []$
  for $i \leftarrow 1$ **to** $|h_t|$ **do**
    $metrics \leftarrow metrics \,||\, \mathcal{M}(n_i^t)$; // Get metrics to sort layer
  $\sigma \leftarrow$ argsort($metrics$)
  $h'_t \leftarrow \sigma(h_t)$; // Sort layer h_t
  for $j \leftarrow 1$ **to** $|h_{t+1}|$ **do**
    $w_{j*}^{t+1} \leftarrow \sigma(w_{j*}^{t+1})$; // Permute next layer weights
  $\mathcal{F} \leftarrow \mathcal{F} \,||\, \text{flatten}(h'_t)$
$\mathcal{F} \leftarrow \mathcal{F} \,||\, \text{flatten}(h_{|f|})$
return $\mathcal{F}$

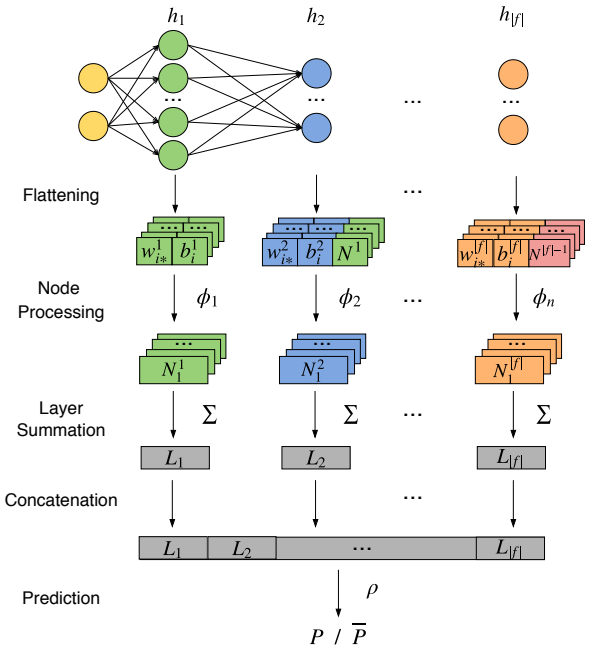


Figure 5: The workflow showing how the meta-classifier processes and learns the set-based representation of a neural network using the DeepSets architecture.

flattened vector of nodes, but instead, as a set of neurons.[2] We could then represent a neural network f as an ordered collection of fully connected layers, where each layer is represented as a set of neurons. Take the neural networks in Figure 4 as an example. Both of them could be represented as a list of two sets (since we do not consider the input layer). The first set contains the three neurons in the hidden layer (i.e., n_1^1, n_2^1 and n_3^1) and the second set contains the single neuron in the output layer (i.e., n_1^2). The key point to note here is that all permutations of any layer will have the same set-based representation, which is crucial to address permutation equivalents.

However, this representation has no merit if it cannot be used to perform property inference. Generic machine learning classifiers rarely operate out of vector-based input representations and cannot effectively interpret and process this set-based representation. In order to process this representation, we need a specialized meta-classifier that can compute and learn functions on sets. For this, we employ the DeepSets architecture formulation [50]. The architecture learns a function of the following form, for a set X :

$$\rho\left(\sum_{x \in X} \phi(x)\right)$$

Recall from Section 2.2, the ϕ function computes an element-level representation for an element x. These element-level representations are summed to obtain the set-level representation, which is passed through the ρ network to get a prediction. The same function ϕ is applied to each element in the set and the summation is unweighted. This allows the function to be permutation invariant and thus is suitable for our task.

In Figure 5, we highlight the workflow of using this specialized meta-classifier to process and learn our set-based representation of the shadow classifier. For a neural network f which has $|f| - 1$ hidden layers and one output layer, the set-based representation of f will correspondingly have $|f|$ sets. The tth set, for instance, represents layer h_t and will comprise of $|h_t|$ elements, i.e., the number of neurons in layer h_t. As we have $|f|$ sets, we will have $|f|$ ϕ networks correspondingly for each layer or set. Since the ρ network is used for the final prediction after obtaining the set-based representation (after summation), we do not use $|f|$ different ρ networks but a single larger ρ network that operates on the set-level representation of all the layers. We use fully connected networks for ϕ_t and ρ.

In essence, the process in Figure 5 can be broken down in the following manner. First, for each node n_i^t in each layer h_t, we *flatten* its weights and biases to a vector. Next, we feed this vector to the corresponding ϕ_t network for the layer to obtain its node-level representation N_i^t. We call this step *node processing*. Note here that Figure 5 shows that the inputs for ϕ_2 and $\phi_{|f|}$ not only include the flattened weights and biases of the nodes but also the concatenated node-level representations of the previous layer. Hence, each ϕ_t, except for ϕ_1, takes as input, for node n_i^t, the weights w_{i*}^t, the bias b_i^t and the context N^{t-1} where

$$N^{t-1} = (N_1^{t-1}, N_2^{t-1}, \ldots, N_{|h_{t-1}|}^{t-1})$$

This is done so that along with the weights and bias, which capture the function performed locally by the neuron, there is also some context for what the node is performing with respect to its inputs. For example, a neuron computing the sum of sums of selected inputs, which it obtains through preceding layers, and another neuron obtaining the sum of differences of selected inputs, which it obtains through preceding layers. Both the neurons locally perform the same function (i.e., summation) but they operate under different contexts. Hence, we provide the node representations N^{t-1} to capture this context of the input provided to the neuron being processed. The node processing step is followed by *layer summation*, where the node representations for each neuron are summed to

[2]Note that it should technically be a multiset instead. The technique we use is applicable to multisets. Additionally, the parameters of a neural network are generally high-precision numbers which means that the likelihood of duplicates is low.

obtain a layer representation L_t:

$$L_t = \sum_i N_i^t$$

Finally, these layer representations are *concatenated* and fed as input to the ρ network for *prediction*, which in our context is P or $\overline{P}$ for the input classifier.

The pseudocode in Algorithm 2 summarizes this set-based approach. As the results show in Section 7, this simple but innovative design performs much better than the baseline, has low memory overhead and requires many fewer shadow classifiers to train.

Algorithm 2: Process and learn the set-based representation.

Input: A neural network f, function ϕ_t for each layer h_t, function ρ
Output: P or $\overline{P}$

$\mathcal{F} \leftarrow []$
$N^0 \leftarrow []$
for $t \leftarrow 1$ **to** $|f|$ **do**
  $N^t \leftarrow []$
  for $i \leftarrow 1$ **to** $|h_t|$ **do**
    $N^t \leftarrow N^t \,||\, \phi_t(w_{i*}^t, b_i^t, N^{t-1})$; // Get node features
  $L_t \leftarrow \sum_i N_i^t$; // Obtain layer features
  $\mathcal{F} \leftarrow \mathcal{F} \,||\, L_t$; // Obtain classifier features
return $\rho(\mathcal{F})$; // Use $\mathcal{F}$ to predict P/ $\overline{P}$

7 EVALUATION

In this section, we first describe the datasets, target models and target properties. We then present the results of our techniques both in effectiveness and efficiency. In the end, we use two case studies to demonstrate the impact of our attack.

7.1 Datasets

US Census Income. The US Census Income Dataset [12] contains census data extracted from the 1994 and 1995 population surveys conducted by the U.S. Census Bureau. This dataset includes 299,285 records with 41 demographic and employment related attributes such as race, gender, education, occupation, marital status and citizenship. The classification task is to predict whether a person earns over $50,000 a year based on the census attributes.

MNIST. The MNIST dataset [22] is a widely used digit recognition dataset. The dataset contains 70,000 handwritten digits with 60,000 samples used for training and 10,000 samples for testing. Each data sample is a 28x28 greyscale image which is size-normalized so that the digits are centered in the images. The classification is a 10-way classification task to recognize the digit shown in the image.

CelebFaces Attributes (CelebA). CelebA [26] is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 binary attribute annotations such as age (young or old), gender, whether the person is wearing a hat, whether the person has wavy hair, etc. Each image has a size of 218x178 pixels. We use this dataset for two classification tasks. One classification task is to detect whether the person is smiling and the other task is to determine the gender of the person.

Hardware Performance Counters (HPCs). Following the data collection techniques developed by Tahir et al. [41], we created our own hardware performance counters datasets which are used to train a model to detect if cryptocurrency mining application is running on the system. We used *perf* with a sampling rate of 2 seconds to profile 15 cryptocurrency mining applications and 19 non-mining applications from the Rodinia [9] and Parboil [40] benchmark suites. All the applications we used are listed in Appendix C. The hardware performance counters data for the applications was collected from a desktop with Intel Core 2 CPU (2.40GHz) running Ubuntu 14.04. For each application, we collected 1500 profiling samples. Each profiling sample contains the values for the same 22 performance counters as used by Tahir et al., such as counts of CPU clock cycles and executed instructions.[3] Overall, each generated dataset contains 36,000 records with 22 attributes.

7.2 Experiment Setup

In the evaluation, we compare our neuron sorting approach (*Sorting*) and set-based representation approach (*Set-Based*) to the *Baseline* approach which uses a flattened vector feature representation. We conducted our experiments on a desktop with Intel Xeon E5-1603 (2.8GHz) CPU and 16GB RAM. The operating system is Ubuntu 16.04. We train all the neural network models using PyTorch [34] on an NVIDIA GeForce GTX 960 GPU. We show the settings of our experiments in Table 1.

Target Models. We train target models for different classification tasks on the datasets described in Section 7.1. For models trained on the HPCs dataset and the US Census dataset, we use neural networks having 3 hidden layers of sizes 32, 16 and 8. For the MNIST dataset, we use neural networks having 3 hidden layers of sizes 128, 32 and 16. For the CelebA dataset, we make use of a pre-trained network, called FaceNet [37], to generate image representations or embeddings of size 512. With FaceNet, we train our own set of fully connected neural networks with 2 hidden layers of sizes 64 and 16, which take as input the embedding instead of the image pixel values. Note that we hold the weights of the pre-trained FaceNet model to be fixed while training our models; i.e., we directly use the embeddings generated by the FaceNet model without updating its weights. In training all our target models, we use the Adam [19] optimizer, ReLu as the activation function, a learning rate of 0.001, a weight decay of 0.01 and 40 maximum epochs of training.

Target Properties. In Table 1, we describe the target properties we want to infer from the target models. For the target models trained on the US Census dataset, the target properties are the disproportion of data samples with some specific attribute values (e.g., gender and race). Similarly, for the CelebA dataset, the target properties are the disproportion of the data samples with some facial attributes. For the MNIST dataset, we want to infer whether the target model was trained using noisy images. To create the noisy images, we add a random brightness jitter to each image. For the HPCs dataset, we want to infer whether the target model was trained on data collected from a machine vulnerable to the Meltdown and Spectre attacks or a machine which had been patched.

[3] Four performance counters are not supported by our system: stalled-cycles-frontend, stalled-cycles-backend, L1-dcache-prefetch-misses and LLC-prefetches.

Table 1: The settings for each experiment, describing the dataset and classification task of the target model, and the target property.

Experiment	Dataset	Target Classifier Task	Target Property (P)	Target Property ($\overline{P}$)
P^1_{Census}	US Census	Binary income prediction	Higher proportion of Women (65% W)	Original distribution (38% W)
P^2_{Census}	US Census	Binary income prediction	Higher proportion of Low Income (80% LI)	Original distribution (50.0% LI)
P^3_{Census}	US Census	Binary income prediction	No whites in the dataset	Original distribution (87% Wh)
P^1_{MNIST}	MNIST	10-way digit classification	Noisy images (with random brightness jitter)	Original images
P^1_{CelebA}	CelebA	Smile prediction	Higher proportion of Attractive faces (68% A)	Original distribution (51% A)
P^2_{CelebA}	CelebA	Smile prediction	Higher proportion of Older faces (37% O)	Original distribution (23% O)
P^3_{CelebA}	CelebA	Smile prediction	Higher proportion of Males (59% M)	Original distribution (42% M)
P^4_{CelebA}	CelebA	Gender classification	Higher proportion of Attractive faces (68% A)	Original distribution (51% A)
P^5_{CelebA}	CelebA	Gender classification	Higher proportion of Older faces (37% O)	Original distribution (23% O)
P^1_{HPCs}	HPCs	Mining activity detection	Data from Meltdown&Spectre vulnerable machine	Data from patched machine

7.3 Attack Effectiveness

Table 2: Accuracy of the property inference attack using different approaches in each experiment.

Experiment	**Baseline** (%)	**Sorting** (%)	**Set-Based** (%)
P^1_{Census}	55.0	89.0	97.0
P^2_{Census}	63.0	85.0	100.0
P^3_{Census}	93.0	100.0	100.0
P^1_{MNIST}	58.0	65.0	85.0
P^1_{CelebA}	67.7	80.2	99.8
P^2_{CelebA}	77.2	91.2	100.0
P^3_{CelebA}	77.2	90.8	100.0
P^4_{CelebA}	73.0	77.5	99.2
P^5_{CelebA}	74.6	84.7	98.8
P^1_{HPCs}	57.0	72.0	88.0

For each experiment in Table 1, we train a set of neural network models that could reveal the target property and a equal-sized set of models that do not. In each experiment, we generate 4,096 models (2,048 with P and 2,048 with $\overline{P}$) as the training set and 512 models (256 with P and 256 with $\overline{P}$) as the test set. Note that, in real life, a model is useful to be shared or released only if it has reasonable quality. Thus, we only generate models which have reasonable performance for their classification tasks. For example, in our experiments, we generate models with more than 90% test accuracy for the MNIST dataset[4] and models with more than 90% test accuracy for both classes on the HPCs dataset.

Since the attacker's goal is to determine whether a given target model could reveal a property or not (i.e., a binary prediction), a useful meta-classifier should have an accuracy higher than 50%, which is a random guess. In Table 2, we show the accuracy of different approaches to infer the target property in each experiment. As we can see from the table, the inference tasks have varying difficulties for the meta-classifier. Except for the P^3_{Census} experiment, the Baseline approach performs badly on most of the experiments, ranging from 55% to 77% in accuracy. Our Sorting approach always outperforms the Baseline approach, demonstrating that permutation equivalence is an important source of complexity for the meta-classifier. We can also see that our Set-Based approach performs even better, with a very high accuracy (ranging from 85% to 100%), and a significant improvement over the baseline, from 55% to 97% in the P^1_{Census} experiment. We believe the excellent performance of the Set-Based approach can also be attributed to the smaller number of parameters in the meta-classifier, which make the classifier easier to train.

In our evaluation, we found that the precision and recall for each experiment are very similar to the accuracy results. Thus, we omit the results for precision and recall in Table 2. We do show the precision and recall results for the P^1_{HPCs} experiment and the P^1_{CelebA} experiment in Section 7.5 and Section 7.6.

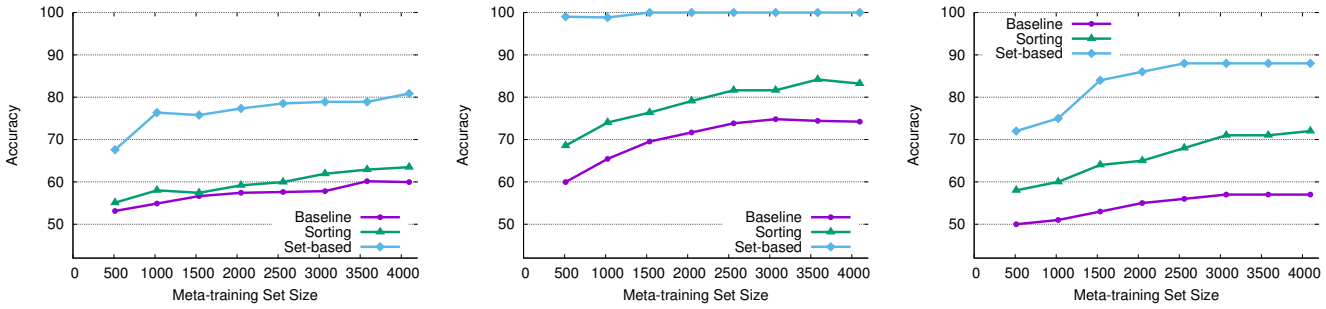
[4]Since we focus on FCNNs, we train FCNNs on the MNIST dataset to demonstrate our attack. Better test accuracy on the dataset can be achieved with other type of ML models.

7.4 Attack Efficiency

In this section, we evaluate the efficiency of the approaches along two different aspects.

Size of the Meta-training Dataset. As described in Section 4, to train a meta-classifier, we first need to train a set of shadow classifiers to build a training dataset for the meta-classifier. In Figure 6, we show the accuracy of the meta-classifier while varying the size of the meta-training dataset. We can see that our Set-Based approach can train a meta-classifier with high accuracy using far less training data than the other two approaches. The Sorting approach and the Baseline approaches require 2 to 7 times more data than the Set-Based approach to train a meta-classifier to its highest accuracy. In inferring properties from large target models, the Set-Based approach can save much time both in training the shadow classifiers and in training the meta-classifier.

Number of Meta-classifier Parameters. In Table 3, we show the number of parameters in the meta-classifiers trained using different architectures. For the Baseline and the Sorting approach, the meta-classifier (*Vector*) is a neural network which takes the vector representation as the input. Thus, the parameters are all the parameters in the neural network. For the Set-Based approach, the meta-classifier (*Set-Based*) uses the DeepSets architecture. The number of parameters for this meta-classifier is the sum of the



(a) Meta-classifier accuracy in P^1_{MNIST}. (b) Meta-classifier accuracy in P^2_{CelebA}. (c) Meta-classifier accuracy in P^1_{HPCs}.

Figure 6: The accuracy of the meta-classifier with varying size of the meta-training dataset.

parameters of all the ϕ_t networks and the ρ network as described in Section 6.2.

Table 3: A comparison of the number of meta-classifier parameters.

Meta-Classifier Type	Census	MNIST	HPCs	CelebA
Vector	$1.3M$	$435.7M$	$1.0M$	$35.3M$
Set-Based	$29.9K$	$270.7K$	$29.7K$	$80.8K$

As we can see from Table 3, the meta-classifier for the Set-Based approach is two or three orders of magnitude smaller in the number of parameters than the other two approaches. When training meta-classifiers, the Set-Based approach requires much less memory and training time than the Sorting approach and the Baseline approach.

7.5 Case study: Inferring Vulnerabilities

In this section, we use the hardware performance counters dataset as a case study to demonstrate how an attacker can use our approach to effectively infer potential vulnerabilities of the neural network model producer.

Hardware performance counters have been used to detect covert cryptomining in clouds and enterprises [41]. Suppose an enterprise trains their covert cryptomining detector using neural networks on the HPCs data collected from their machines, which do not have patches for the Meltdown and Spectre vulnerabilities applied. Since these patches have a notable performance impact, we would expect this fact to affect the performance counter data (indeed, performance counters have been proposed to detect Meltdown and Spectre attacks [33, 44]) and thus influence the trained model. The enterprise may share their covert cryptomining detector with other parties. An adversary who gets the access to the model could analyze it to infer whether the company's machines are vulnerable to Meltdown and Spectre or not. With such information, the adversary could perform follow-up attacks against the enterprise.

To demonstrate this attack, we first create a HPCs dataset (*unpatched*) as described in Section 7.1 on a machine that is vulnerable to Meltdown and Spectre (i.e., the operating system of the machine has not installed the patches for Meltdown and Spectre). The operating system running on the machine is Ubuntu 14.04. Then we create another HPCs dataset (*patched*) on the same machine after the patches have been installed. We train 2048 cryptomining detectors on the unpatched HPCs dataset and another 2048 detectors on the patched dataset. Thus, we have the ground truth about the vulnerability of the training machine. We train meta-classifiers using different approaches and compare their effectiveness. As shown in Table 2, the Baseline approach only has 57% accuracy in predicting whether the model producer's machine is vulnerable to Meltdown and Spectre. Our Set-Based approach, however, achieves 88% accuracy in inferring the vulnerabilities. We show the results of precision and recall in Table 4.

Table 4: Precision and recall results of different approaches in experiment P^1_{HPCs}. P.: Precision; R.: Recall.

	Baseline		Sorting		Set-Based	
	P	R	P	R	P	R
Vulnerable	0.56	0.58	0.71	0.70	0.88	0.87
Not Vulnerable	0.57	0.55	0.71	0.71	0.89	0.90

If the model producer of the cryptomining detector releases the data collection settings (e.g., CPU type and operating system) along with the model, then the attacker can just collect his HPCs dataset using the same configuration. Otherwise, the attacker can enumerate possible configurations and create HPCs dataset for each of the configuration. But this is still manageable as there are limited type of active CPUs and OSes in the market. To simulate this scenario, we create another HPCs dataset from another machine with the same configuration as the one used in Section 7.1 to train the shadow classifiers. The accuracies of the three approaches are shown in Table 5. Our Set-Based approach can still achieve very high accuracy (86%).

Table 5: Accuracy of the property inference attack on non-overlapping datasets.

Experiment	Baseline (%)	Sorting (%)	Set-Based (%)
P^1_{HPCs}	56.0	71.0	86.0
P^1_{CelebA}	62.7	73.4	96.3

7.6 Case study: Inferring Training Data Distribution

The distribution of the training data, in some cases, is also confidential information. For example, a classifier producer could infer certain hidden distribution of their competitor's training data to

uncover the secret "sauce" to build more effective classifiers [5]. One can also infer the training data distribution from the released model to check if it confirms with the model producer's claim about the training set. In this section, we demonstrate how our approach can improve the effectiveness in discovering disproportionate data distributions for neural networks trained for more complex classification tasks.

Face attribute prediction (e.g., smile detection or gender classification) is a common machine learning task. In this case study, we consider a model producer who releases their neural network models for smile detection and for gender classification. As described in Table 1, these models are trained on datasets that either have higher proportions of attractive faces or older faces. We evaluate the effectiveness of different approaches in discovering such disparities.

However, facial attribute prediction is a very complex task which requires a large set of face images and a large convolutional neural network to achieve good performance. Training such models from scratch requires a large amount of computational resources while the trained models often give very small, if any, performance boosts over the existing pre-trained models. For this reason, it is now a common technique to leverage pre-trained models, such as the Facenet model [37] or the Deepface model [42], to train one's customized classification tasks. In this case study, we consider a model producer who uses pre-trained models to create their models for smile detection and gender classification.

As described in Section 7.2, we train the target models using a pre-trained FaceNet model with its weights held fixed and use the 512-dimensional embeddings generated from it to train a 3-layer neural network. We train the shadow classifiers using the same structure and we use the parameters in the 3-layer neural network as the feature representation in our evaluation. As shown in Table 2, even though the Baseline approach has reasonable performance (around 75% accuracy), our Set-Based approach always achieves near-perfect accuracy in discovering the changed proportions in the training dataset. We show the results of precision and recall in Table 6. Besides, as shown in Figure 6, our Set-Based approach achieves near-perfect accuracy while requiring only about **1/3** as many shadow classifiers as the Baseline approach. This would save a great amount of time and resources in performing the inference tasks as compared with the Baseline approach. Our results show that our approach is very effective even when the adversary is privy only to some of the layer's parameters and knowledge of the underlying pre-trained network used by the model producer. We also simulate the scenario in which the adversary only has part of the CelebA dataset that is non-overlapping with the dataset used by the model producer. Similarly, as shown in Table 5, our Set-Based approach can still achieve very high accuracy.

It is interesting to note that disproportionate ratios in seemingly unrelated attributes such as attractiveness are embedded and can be discovered with such accuracy in neural network models trained to classify a person's gender or detect whether a person is smiling.

Table 6: Precision and recall results of different approaches in experiment P^1_{CelebA}. P.: Precision; R.: Recall.

	Baseline		Sorting		Set-Based	
	P	R	P	R	P	R
Attractive	0.66	0.72	0.83	0.76	1.00	0.99
Unattractive	0.69	0.63	0.78	0.84	0.99	1.00

8 DISCUSSION

8.1 Limitations and Future Work

Other Types of Neural Networks. In this paper, we mainly focus on fully connected networks. However, fully connected (FC) layers are widely used in all sorts of neural networks such as convolutional neural networks. In Section 7.6, we show that we can perform property inference just using the FC layers trained on the output of a pre-trained convolutional model. We believe it is possible to apply our approach to other types of neural networks and to find some forms of equivalents in other types of computational layers. We plan to investigate property inference attacks against other types of computational layers in future work.

Overfitting. As studied in [38], overfitting is a major factor that causes a model to be vulnerable to membership inference attacks. However, it is not clear whether overfitting of target classifiers has any role in the property inference attack. In our case, as discussed in Section 7.3, all our trained models have good generalizability, as measured by the accuracy on the test sets. We plan to study the relationship between overfitting and property inference in more detail in the future.

Membership Inference. Membership inference and property inference are two different but related problems. One may propose to use membership inference to infer all the members in the training dataset and then use them to infer the properties. However, current state of art for membership inference usually only finds some members with a relatively high degree of uncertainty. It is also not practical to infer all the members in a large dataset such as CelebA. Property inference, instead, focuses on the property directly. We plan to study the relationship between membership inference and property inference in future work.

Multi-label or Regression based Properties. We currently only study binary-class property inference. However, a more powerful attack would entail predicting from multiple classes, such as inferring which operating system a model provider of a mining detector is using, or performing a regression task, such as predicting the ratio of genders used to train a smile detector. Our preliminary results show promise for our approach to be extended for multi-class and regression tasks but we leave detailed analysis for future work.

Generating Training Data for Shadow Models. To train the meta-classifier, the attacker first needs training data to train the shadow classifiers. Although generating the training data is not the focus of this work, there are many existing approaches to facilitate the generation of training data. For example, if the attacker does not have access to similar training dataset (e.g., public accessible datasets like CelebA) or knowledge of the dataset generation technique, he could generate synthetic training data for the target model using model-based synthesis [38]. We leave experiments using this technique of data generation for future work.

8.2 Alternative Approaches

In addition to neuron sorting and set-based representation, we also briefly investigate several other alternative approaches. We discuss some of our exploration of these alternative approaches.

Augmentation. In machine learning, data augmentation is a common strategy for improving generalizability of ML models. The training dataset is expanded with new data points generated using deterministic or randomized transformations. For example, data augmentation for images could be achieved by adding rotated or noisy images to the dataset. Similarly, in our case, to deal with node permutation equivalence, we could generate different permutation equivalents to augment the meta-classifier training dataset so that it generalizes across these equivalents. However, we find that this approach does not work well in practice. It only has marginal improvement over the baseline approach. One reason is that, as discussed in Section 5, the number of permutation equivalents is superexponential and it is therefore impractical to generate all the permuted versions of the classifiers to cover the huge search space.

Graph-based Representation. As we can see from Figure 2 and Figure 5, a neural network is often visualized as a graph. Thus, it is more intuitive to represent a neural network as a graph where all neurons from all layers (including the input and output layer) are treated as nodes with directed edges pointing to the nodes in the next layer. The weight of the edge between two nodes in the graph is derived from the weights connecting the two corresponding neurons in the neural network. Similar to the case for set-based representations, the meta-classifier needs to be able to process and interpret the proposed graph structure. We experimented with building a meta-classifier using the Graph Convolutional Network (GCN) proposed by Kipf et al. [20]. However, we find this approach has mixed performance: it worked well for a few selected inference tasks and had poor performance for others. Additionally, it takes much more time and resources to train the meta-classifier compared to our other proposed approaches. We believe that the poor performance for the inference tasks is because the GCN architecture does not address permutation equivalents directly which take the form of isomorphisms in the graph-based representation. GCNs may also be a poor fit for this task as they were designed for analysis of social networks, which have a much sparser and less regular structure than deep neural networks.

Accuracy-based Classification. It is reasonable to expect that a classifier trained on a dataset with property P will have a better performance on test datasets that also have property P compared to those that have property $\overline{P}$ (indeed, this is a frequently voiced concern regarding the fairness of machine learning algorithms trained on biased data sets). This suggests that a strategy to infer a property of the training dataset by analyzing the target classifier's performance on test datasets that variously have and do not have the property P. One advantage of such strategy would be that it could be applied in a black-box setting. To investigate this, we trained 2048 classifiers on the noisy version of the MNIST data set and 2048 on the clean version (as described in Section 7.1). We then evaluated the performance of each classifier on a clean and a noisy test set, plotting the results in Figure 7. We can observe that there is a slight trend for classifiers trained with a property P to perform better on property P; however, the difference between the two distributions is very marginal.

To further investigate how well this method could be used for property inference, we trained a k-nearest neighbor (kNN) meta-classifier using these 4096 2-dimensional points as a labeled training set. We then trained 256 more classifiers each on noisy and clean

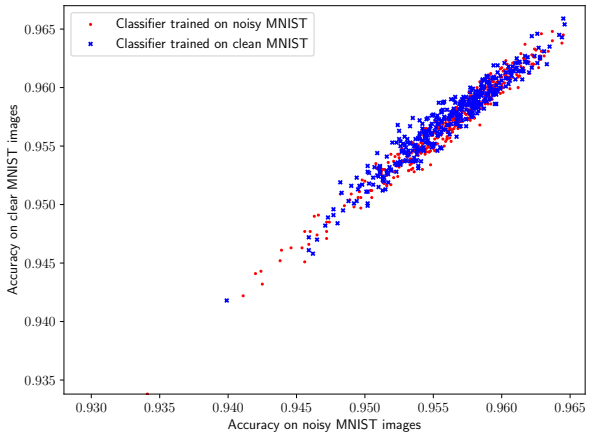


Figure 7: Comparing the accuracy on noisy (x axis) and clean (y axis) test sets of classifiers trained on noisy (red) or clean (blue) training sets. While there is a slight trend for classifiers trained on clean data to perform marginally better on clean data, and vice versa, there is no strong separation between the two types of classifier.

MNIST training sets, respectively, and evaluated the performance of the kNN meta-classifier at property inference. We repeated the same experiment with several properties, comparing the results to our Sorting and Set-Based approaches. The results, shown in Table 7, show that the accuracy-based kNN classifier does not even consistently outperform the baseline, and in general performs worse than either of our approaches. It is possible, however, that the accuracy of a classifier on sets with and without a property could be a useful *additional* input to a classifier that also uses the model parameters; we plan to study this in future work.

Table 7: Accuracy of the property inference attack using kNN on classifier accuracies compared to other approaches.

Experiment	Baseline (%)	Sorting (%)	Set-Based (%)	kNN (%)
P^1_{Census}	55.0	89.0	97.0	60.0
P^2_{Census}	63.0	85.0	100.0	84.0
P^3_{Census}	93.0	100.0	100.0	75.0
P^1_{MNIST}	58.0	65.0	85.0	56.0

8.3 Possible Defenses

Property inference attacks leverage the similarity in models' parameters to predict if a target model could reveal some property. To defend against such an attack, the model producer could manipulate the parameters of the model directly or indirectly to make it "different" from the shadow models that an attacker uses to train the meta-classifier. We discuss possible defenses against property inference attacks which are based on this principle.

Using Node Multiplicative Transformations. One of our observations is that, for a neuron with an activation function of ReLU or LeakyReLU, multiplying the weights and bias of the neuron by

some constant and dividing the weights connecting it to the next layer by the same constant will result in the same output of the neural network. Thus, we could randomly pick some neurons in each layer and then perform this transformation using a random constant to each of the selected neurons. We tested this idea with the CelebA dataset. The results show that the attack accuracy of our approaches decreases as we increase the fraction of perturbed neurons. However, this defense can only be used by deep neural networks that use ReLU or LeakyReLU as activation function.

Adding Noisy Data to the Training Set. Another possible approach to defend against this attack is to add some noisy data to the training set. The model producer could add noise by flipping the labels of some training samples. This will affect the model parameters which increases the difficulty of the property inference task. In our HPCs dataset experiment, we randomly flipped the labels of 5% of the training data for the target classifiers. While this caused the test accuracy of the target classifiers to drop to 90%, the inference accuracy of our set-based approach dropped significantly from 88% to 73%. However, this defense is unlikely to be deployed by model producers as adding noisy data might hamper the effectiveness and consequently, the utility of the model.

Encoding Arbitrary Information. Deep neural networks have huge capacity for "memorizing" arbitrary information [51]. Song et al. [39] show that they are able to encode a significant amount of information about the training set in a neural network model while maintaining the model's quality and generalizability. The attack involves updating the model parameters to remember this extra information. While the attack followed a different setting than ours, we think it could be employed instead as a potential defense against our attack. Model producers could use their techniques to encode some arbitrary information in the model, making the parameters look different from those of the shadow classifiers, possibly hampering the meta-classification task.

9 RELATED WORK

Assorted inference attacks have previously been launched on machine learning models with varying degrees of success. Ateniese et al. [5] first proposed the concept of a property inference attack, which they demonstrated against SVM and HMM models. Our experiments show that the direct application of their techniques does not succeed when using neural networks, but our methods for permutation invariant representations make the attack effective in practice. Apart from property inference, there are a number of other privacy threats against ML models.

Membership inference attacks against ML models aim to infer whether a specific data record was in or out of the target model's training dataset. Similar to our attack strategy, Shokri et al. [38] train multiple shadow models to help train an attack classifier, which ultimately is used to perform membership inference for the target model. Hayes et al. [18] use generative adversarial networks to perform membership inference attacks against generative models. However, membership inference focuses on the privacy of an individual record, while the concept of information leakage in our work is more general. It concerns any type of confidential information revealed about the training dataset beyond the predictions the model is intended to perform.

Model inversion attacks aim to infer the missing information for a data record using an ML model and known incomplete information about the data record. Fredrikson et al. [16] invert a linear regression model to infer patients' genomic markers given their corresponding output from the model and auxiliary demographic information about them. In their following work [15], they invert decision trees and neural networks using predicted confidence values to leak confidential information about a data record. Similar to the membership inference attacks, this type of attack also focuses on individual records. Recent work by Yeom et al. [49] show that membership inference attacks and model inversion attacks are deeply related and are both sensitive to overfitting of target models.

Model extraction attacks seek to obtain the parameters of a black-box ML model given the outputs it predicts for a chosen set of inputs. Tramèr et al. [43] demonstrate that they can successfully extract popular model types including logistic regressions, SVMs, and deep neural networks, against production MLaaS providers. Oh et al. [31] show that they can successfully reverse-engineer internal information such as the architecture and hyperparameters of black-box neural network models. These attacks can be a stepping stone for our attack. For example, an adversary could use model extraction to obtain a near-equivalent model of the target black-box model, then perform property inference on the white-box model to infer confidential properties. Wang et al. [46] propose a framework to steal the hyperparameters in the objective functions of different machine learning algorithms including neural networks. If we treat the hyperparameter for a neural network, such as mini-batch size, as a property of the training process, our approach can also be used to infer the hyperparameter.

Beyond data inference, ML models have been shown to be subject to a variety of other attacks, such as, adversarial attacks on object detectors [27] or classifiers [13, 29, 30, 32], malicious algorithms which discreetly memorize the training data samples [39], and malicious classifiers which mislead the classifier to misbehave in presence of an input trigger [25]. Our property inference attack on deep neural networks adds to this growing set of risks that users of machine learning must consider.

10 CONCLUSION

We considered the problem of a property inference attack on fully connected neural network models, and developed a new concept—permutation invariance—to address the complexity of this task. Specifically, we developed two approaches leveraging permutation invariance, neuron sorting and set-based representation, to infer global properties of the training datasets that the model producer did not intend to share. We showed that our approaches are effective at inferring various data properties on several real-world datasets. We also showed the practical impact of our work through two case studies: identifying whether machines used to train a cryptomining detector model were patched for a vulnerability, and detecting unbalanced training dataset distributions.

We also identified a number of directions for future work, including extending our work to neural network layers that are

not fully connected, performing multi-class and regression meta-classification tasks, and developing countermeasures to the property inference attacks. We also believe that the permutation invariance property may have other applications beyond property inference, perhaps in quality assurance of FCNN models.

ACKNOWLEDGEMENTS

This work was supported in part by NSF CNS 13-30491, NSF CNS 14-08944, and DOE 0E0000780. The views expressed are those of the authors only. We appreciated valuable insights from our CCS reviewers and our shepherd, Giuseppe Ateniese.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/2976749.2978318

[2] Amazon. 2018. Machine Learning at AWS. Retrieved August 15, 2018 from https://aws.amazon.com/machine-learning/

[3] Android Developers. 2018. Android Neural Networks API. Retrieved August 15, 2018 from https://developer.android.com/ndk/guides/neuralnetworks/

[4] Apple Inc. 2018. Core ML. Retrieved August 15, 2018 from https://developer.apple.com/documentation/coreml

[5] Giuseppe Ateniese, Luigi V Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. 2015. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *International Journal of Security and Networks* 10, 3 (2015), 137–150.

[6] BigML Inc. 2018. BigML. Retrieved August 15, 2018 from https://bigml.com/

[7] Joy Buolamwini and Timnit Gebru. 2018. Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. In *Proceedings of the 1st Conference on Fairness, Accountability and Transparency (Proceedings of Machine Learning Research)*, Sorelle A. Friedler and Christo Wilson (Eds.), Vol. 81. PMLR, New York, NY, USA, 77–91. http://proceedings.mlr.press/v81/buolamwini18a.html

[8] Caffe. 2018. Caffe Model Zoo. Retrieved August 15, 2018 from http://caffe.berkeleyvision.org/model_zoo.html

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC 2009)*. 44–54.

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* 49 (2016), 1162–1174.

[11] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 559–570.

[12] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[13] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2017. Robust physical-world attacks on machine learning models. *arXiv preprint arXiv:1707.08945* (2017).

[14] Eric Florenzano. 2016. Gadientzoo. Retrieved August 15, 2018 from https://www.gradientzoo.com/

[15] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1322–1333.

[16] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security Symposium*. 17–32.

[17] Google Cloud. 2018. Cloud Machine Learning Engine. Retrieved August 15, 2018 from https://cloud.google.com/ml-engine/

[18] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. 2017. LOGAN: evaluating privacy leakage of generative models using generative adversarial networks. *arXiv preprint arXiv:1705.07663* (2017).

[19] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[20] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[22] Yann LeCun, Corinna Cortes, and Christopher J Burges. 2018. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[23] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 5.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[25] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning Attack on Neural Networks. In *25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221, 2018*. The Internet Society.

[26] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2018. Large-scale CelebFaces Attributes (CelebA) Dataset. Retrieved August 15, 2018 from http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

[27] Jiajun Lu, Hussein Sibai, and Evan Fabry. 2017. Adversarial Examples that Fool Detectors. *arXiv preprint arXiv:1712.02494* (2017).

[28] Microsoft. 2018. Azure Machine Learning. Retrieved August 15, 2018 from https://azure.microsoft.com/en-us/services/machine-learning-studio/

[29] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[30] Seyed Mohsen Moosavi Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[31] Seong Joon Oh, Max Augustin, Bernt Schiele, and Mario Fritz. 2018. Towards Reverse-Engineering Black-Box Neural Networks. *International Conference on Learning Representations* (2018).

[32] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 506–519.

[33] Cody Pierce. 2018. Detecting Spectre And Meltdown Using Hardware Performance Counters. Retrieved August 15, 2018 from https://www.endgame.com/blog/technical-blog/detecting-spectre-and-meltdown-using-hardware-performance-counters

[34] PyTorch core team. 2018. Pytorch. Retrieved August 15, 2018 from http://pytorch.org/

[35] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

[36] F. Rosenblatt. 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review* (1958), 65–386.

[37] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[38] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18.

[39] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Machine Learning Models that Remember Too Much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 587–601.

[40] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[41] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. 2017. Mining on Someone Else's Dime: Mitigating Covert Mining Operations in Clouds and Enterprises. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 287–310.

[42] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1701–1708.

[43] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*. 601–618.

[44] Trend Micro. 2018. Detecting Attacks that Exploit Meltdown and Spectre with Performance Counters. Retrieved August 15, 2018 from https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/

[45] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2014. OpenML: networked science in machine learning. *CoRR* abs/1407.7722 (2014).

[46] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing Hyperparameters in Machine Learning. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[47] Xueyang Wang and Ramesh Karri. 2013. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 79.
[48] Perf Wiki. 2018. Linux profiling with performance counters. Retrieved August 15, 2018 from https://perf.wiki.kernel.org/index.php/Main_Page
[49] Samuel Yeom, Matt Fredrikson, and Somesh Jha. 2017. The Unintended Consequences of Overfitting: Training Data Inference Attacks. *arXiv preprint arXiv:1709.01604* (2017).
[50] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. In *Advances in Neural Information Processing Systems*. 3394–3404.
[51] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2017. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations (ICLR)*.

A META-TRAINING ALGORITHM

In Algorithm 3, we show our algorithm to train the meta-classifier.

Algorithm 3: Training of the meta-classifier.

Input: An array of training sets $\mathcal{D}$, an array of labels l, an array of training configurations E
Output: The trained meta-classifier

$D_{\mathcal{MC}} \leftarrow \{\emptyset\}$
for $i \leftarrow 1$ **to** $|\mathcal{D}|$ **do**
  $f_i \leftarrow \text{train}(\mathcal{D}_i, E_i)$
  $\mathcal{F}_i \leftarrow \text{getFeatureRepresentation}(f_i)$
  $D_{\mathcal{MC}} \leftarrow D_{\mathcal{MC}} \cup \{\mathcal{F}_i, l_i\}$
$\mathcal{MC} \leftarrow \text{train}(D_{\mathcal{MC}})$
return $\mathcal{MC}$

B PROOF OF PERMUTATION EQUIVALENCE

We first describe the notations we use for the proof. For simplicity, we denote the output computed by a node n_i^t on input x as $n_i^t(x)$. Likewise, we denote the output computed by layer h_t as $h_t(x)$. That is:

$$h_t(x) = (n_1^t(x), n_2^t(x), \ldots, n_{|h_t|}^t(x))$$

Now, a node permutation on layer h_t involves permutation of the neurons in layer h_t to obtain $\sigma(h_t)$ and permuting the weights of each neuron in layer h_{t+1} so the weights of node n_i^{t+1} are now $\sigma(w_{i*}^{t+1})$. For ease of notation, we denote the outputs of this transformed layer $\sigma(h_t)$ for input x to be $\sigma(h_t)(x)$ and likewise, we denote the output of the neuron n_i^{t+1} after permutation of its weights to be $\sigma(n_i^{t+1})$

In order to prove Proposition 5.1, it would suffice to show the following: *For any node permutation applied on layer h_t, the output of layer h_{t+1} remains unchanged for any input x*. Since node permutation does not affect any of the layers before h_t, we can consider the input to h_t to be the same regardless of the permutation. Additionally, since node permutation does not affect the layers after h_{t+1}, we can rest assured that the output of the neural network remains the same as long as the output of layer h_{t+1} remains unchanged. As the order of nodes in layer h_{t+1} does not change after node permutation, we only need to prove that the output of every node in layer h_{t+1} remains unchanged for any input x.

Hence, we need to prove that, for all x,

$$n_i^{t+1}(h_t(x)) = \sigma(n_i^{t+1})(\sigma(h_t(x)))$$

It can be proved with the following equations:

$$\begin{aligned} n_i^{t+1}(h_t(x)) &= \gamma(w_{i*}^{t+1} \cdot h_t(x) + b_i^{t+1}) \\ &= \gamma((\sum_j w_{ij}^{t+1} n_j^t(x)) + b_i^{t+1}) \\ &= \gamma((\sum_j w_{i\sigma(j)}^{t+1} n_{\sigma(j)}^t(x)) + b_i^{t+1}) \\ &= \sigma(n_i^{t+1})(\sigma(h_t)(x)) \end{aligned}$$

C HARDWARE PERFORMANCE COUNTERS DATASETS

In Table 8, we list the cryptocurrency mining applications we profiled in creating our hardware performance counters datasets. In total, we profiled 13 different cryptocurrencies with 13 different proof-of-work (PoW) algorithms.

Table 8: The cryptocurrency mining applications we profiled along with their PoW algorithms.

Cryptocurrency	Proof-of-Work Algorithm
Litecoin	Scrypt
Bitcoin	SHA256
Bytecoin	CryptoNight
Dashcoin	CryptoNight
QuazarCoin	CryptoNight
VertCoin	Lyra2rev2
FeatherCoin	Neoscrypt
Dashcoin	X11
Auroracoin	Qubit
Myridacoin	Yescrypt
Digibyte	Skein
Digibyte	Myr-gr
Groestlcoin	Groestl
Maxcoin	Keccak
Zcoin	Lyra2z

In Table 9, we list the non-mining applications we profiled in creating our hardware performance counters datasets.

Table 9: Non-mining applications we profiled along with the benchmark suite they belong to.

Application	Benchmark	Application	Benchmark
stencil	Parboil	tpacf	Parboil
lbm	Parboil	cutcp	Parboil
histo	Parboil	mri-q	Parboil
backprop	Rodinia	euler3d_cpu	Rodinia
particle_filter	Rodinia	pathfinder	Rodinia
datagen	Rodinia	heartwall	Rodinia
kmeans	Rodinia	lavaMD	Rodinia
leukocyte	Rodinia	lud_omp	Rodinia
srad_v1	Rodinia	srad_v2	Rodinia
sc_omp	Rodinia		