DNNShield: Embedding Identifiers for Deep Neural Network Ownership Verification

Jasper Stang
University of Würzburg
jasper.stang@uni-wuerzburg.de

Torsten Krauß
University of Würzburg
torsten.krauss@uni-wuerzburg.de

Alexandra Dmitrienko
University of Würzburg
alexandra.dmitrienko@uni-wuerzburg.de

The surge in popularity of machine learning (ML) has driven significant investments in training Deep Neural Networks (DNNs). However, these models that require resource-intensive training are vulnerable to theft and unauthorized use. This paper addresses this challenge by introducing DNNShield, a novel approach for DNN protection that integrates seamlessly before training. DNNShield embeds unique identifiers within the model architecture using specialized protection layers. These layers enable secure training and deployment while offering high resilience against various attacks, including fine-tuning, pruning, and adaptive adversarial attacks. Notably, our approach achieves this security with minimal performance and computational overhead (less than 5% runtime increase). We validate the effectiveness and efficiency of DNNShield through extensive evaluations across three datasets and four model architectures. This practical solution empowers developers to protect their DNNs and intellectual property rights.

I. Introduction

Machine learning (ML) has become ubiquitous in our daily lives, driving significant advancements in fields, e.g., speech recognition [16], object detection [33], [30], [24], [26], natural language processing [12], and predictive analysis [3]. During model training, ML extracts generalized patterns from training data. Those patterns, represented by model parameters, are then used to generate predictions for unseen data. Recently, larger ML models, so-called Deep Neural Networks (DNN), with larger learning capacities providing more accurate predictions emerged. However, training DNNs necessitates extensive computational resources [45] and leverages proprietary datasets. Hence, DNNs are valuable assets for model creators, requiring intellectual property (IP) protection measures. Related IP protection methods are Watermarking and Passporting.

Existing DNN IP Protection Methods. Watermarking [47], [31], [43], [50], [51], [14], [34], [2], [54], [32], [21], [27] involves embedding stealthy but identifiable markers as reference watermarks within the model during training. The watermarks (identifiers) can be extracted from a previously marked model using a confidential method (also referred to as key). The extracted identifiers serve as evidence of ownership. In case of suspected model copyright infringement, the watermark extracted from the suspected model (using the secret key) can be compared to the model's reference watermark, proving the copyright infringement. However, many DNN watermarking techniques [47], [31], [43], [50], [51], [14], [34], [54], [32], [21], [27] require key secrecy within their threat model. If ownership is proven by disclosing the extraction key and watermark publicly, subsequent ownership watermark verification is not legitimate because the key is now accessible to anyone who observed the initial extraction. Another method [2] does not require key disclosure but relies on manipulating the dataset, which negatively impacts model performance.

Passporting [19], [20] was also proposed, which embeds extra trainable "Passporting" layers within the model architecture, essentially entangling model and Passporting parameters. Removing the trained Passporting layers degrades the model's prediction performance. Therefore, a model can only be efficiently utilized with the untouched Passporting layers which serve as a means for model identification. However, existing Passporting schemes have four drawbacks: First, the trainable Passporting layers introduce additional learning overhead. Secondly, it has been shown to be prone to attacks [11], where the Passporting layers are substituted with new layers that mimic their functionality. Third, the proposed method is limited to being used in conjunction with convolutional layers, thus, making it applicable to only a limited set of model architectures. Finally, the system does not adequately protect unfinished training stages with decent prediction performance, known as checkpoints, as the Passporting layers (identifier) are continuously modified throughout the training process. This presents a significant security risk for models with lengthy training durations, as intermediate checkpoints are left unprotected.

Summarized, Watermarking fundamentally relies on key secrecy allowing only for a single public ownership verification or manipulates the dataset which degrades model performance. Alternatively, Passporting introduces additional training overhead, is vulnerable to adversarial attacks, is restricted to a single type of layer, and cannot protect model checkpoints.

Approach. To address the limitations of existing DNN IP protection methods, we propose DNNShield , a novel approach that integrates publicly known protection layers into the model architecture. The layers are untrainable, and hence remain unchanged during training. The approach seamlessly integrates

two types of Protection layers into the model architecture, namely Hadamard and Permutation layers allowing for usage in conjunction with both convolutional and linear layers. Similar to Passporting, the idea behind protection layers is that the data flow through the model is altered, such that the regular model layers are entangled with the Protection layers. As a consequence, an adversary is faced with the dilemma of either leaving the Protection layers unchanged, which allows for reliable model ownership verification or manipulating the Protection layers, resulting in a significant reduction in model performance due to the entanglement of Protection layers with the regular model parameters.

Contributions. In particular, we make the following contributions:

- We propose DNNShield , a novel DNN protection method that eliminates the reliance on secret keys, allowing for repeated public ownership claims, while maintaining negligible impact on model performance and training time.
- DNNShield relies on integrating untrainable protection layers into the model architecture, that enable ownership claims. The layers result in negligible training overhead as DNNShield does not introduce additional trainable parameters. Further, as the layers do not change during training intermediate checkpoints and the final model are secured.
- We present two types of untrainable Protection layers, namely Hadamard and Permutation layers, that can be seamlessly integrated into various architectures. The layers can be used in conjunction with convolutional or linear layers, hence DNNShield is applicable to most architectures.
- We evaluate DNNShield in diverse application scenarios, in particular the approach is evaluated with four model architectures, namely ResNet-18 [24], a Convolutional, a Fully Connected model, and a Vision Transformer [17]. Further, it is evaluated on three datasets: MNIST [15], CIFAR-10 [29], and GTSRB [41]. Moreover, the robustness against third-party manipulation including adaptive adversarial attacks is shown. DNNShield has negligible impact on the model performance and a runtime overhead of less than 5% for Hadamard layers in our experiments.

In summary, this work introduces a novel and efficient DNN IP protection method, offering a robust defense against various attacks. DNNShield adds novel protection layers into the architecture, that do not rely on secrecy and are untrainable, hence, do not introduce training overhead. As the layers remain unchanged during training, both, intermediate checkpoints and the final model, are secured. Further, they can be used in conjunction with convolutional or linear layers, making them applicable to most modern model architectures. Overall, DNNShield addresses the limitations of existing DNN IP protection approaches while providing an intuitive and robust ownership verification.

Outline. We provide background information in Sect. II

and depict the considered scenario and threat model in Sect. III. Sect. IV details the approach of DNNShield followed by a security analysis in Sect. V. The evaluation results are reported in Sect. VI and Sect. VII elaborates on additional considerations. Finally, related works are discussed in Sect. VIII before we draw a conclusion in Sect. IX.

II. BACKGROUND

Below, we provide information on data representations, DNN layers, and important metrics necessary for understanding our approach, as well as common attacks on DNN IP protection.

A. Data Representations

Matrices are fundamental components in ML used to represent the data on which models are trained and run. Matrices have dimensionalities ranging from one (referred to as a vector) to multiple dimensions. For image data, the representation usually adheres to a three-dimensional arrangement: c, w, and h, where c initially indicates the three color channels red, green, and blue, while w and h represent the width and height of the image. Certain layers, such as convolutional layers, introduce additional channels, potentially expanding the representation beyond the initial three channels. Unlike convolutional layers, Fully-Connected layers typically abstract away the spatial dimensions, aligning the input along the first dimension. This knowledge is crucial as our approach operates on matrices.

B. Neural Network Layers

To comprehend our approach, it is essential to have a good understanding of certain model layers. Specifically, our approach works in conjunction with linear and convolutional layers. Hence, it is crucial to understand their inner workings.

Fully-Connected Layer. Those layers are commonly used in most model architectures [24], [30], [26] and perform the calculation $y = x \cdot w^T + b$. The learnable parameters, namely the weights and bias matrices, are represented by w and b, while x and y denote the input and output matrices and $\cdot$ is the dot product operation. The transposed operation (T) flips the matrix over its diagonal.

Convolutional Layer. Fully-Connected (FC) layers, which have a large number of parameters and perform computationally intensive dot product calculations, have limited applicability to image processing. To overcome this challenge, convolutional layers have emerged as an alternative, especially in DNN architectures such as ResNet [24]. Unlike FC layers, which process information over the input simultaneously, convolutional layers analyze the input data in a fragmented and sequential manner. They operate on small receptive fields, which are essentially windows that scan the input image. As the receptive field moves across the image, a filter, also called kernel, which is a small matrix of trainable weights, is applied to each receptive field, effectively extracting relevant features from the data. This sequential approach allows convolutional layers to capture spatial relationships and patterns efficiently.

The underlying mathematical method, called cross-correlation, can be used to process multiple input values and produce a more compact output. The number of trainable parameters is significantly reduced as the kernel only contains a fraction of the weights of a FC layer. The input vector, x, undergoes a cross-correlation operation denoted by $*$. The resulting output vector, y, is calculated using $y = (x * f)$, where f denotes the filter (kernel) used by the convolutional layer. In DNNs a single convolutional layer typically incorporates multiple kernels [24], [26], [30], each producing an output feature map (w, h) that contributes to a distinct channel dimension c.

Element-Wise Multiplication Layer. The element-wise multiplication [13] takes a matrix and outputs a new matrix with identical dimensions, performing element-wise multiplication for each value. The layer calculates $y = x \circ k$. Here, x refers to the input matrix and k refers to the matrix by which x is multiplied. The symbol $\circ$ denotes the element-wise multiplication. Our approach utilizes those layers to embed unique identifiers into the model architecture.

C. Metrics

In the following, we first introduce cosine similarity, which is used for ownership verification. Then, we introduce the accuracy metric used to evaluate the performance of DNNs.

Cosine Similarity. Cosine similarity is a fundamental metric for analyzing data, measuring the similarity between two vectors. It is used, for instance, to measure similarity of ML models for backdoor detection [28], [38]. It is calculated by dividing the dot product of two vectors by the product of their magnitudes. Essentially, the alignment of two vectors within an inner product space are measured which yields values that range from -1 to 1. The formula to calculate the cosine similarity is $cos = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}$, with x_1 and x_2 being two vectors. If the cosine similarity value is 1, this implies that the vectors are identical and perfectly aligned. Conversely, if the value is -1, the vectors point in opposite directions, indicating complete dissimilarity. Additionally, if the cosine similarity value equals 0, it means the vectors are orthogonal and lack directional alignment. Our approach is based on measuring the similarity between the parameters of two untrainable (fixed parameters) protection layers to determine if one model closely resembles the other.

Model Performance Metrics. The accuracy metric assesses a model's predictive capabilities, essentially expressing the ratio of correct predictions for a set of input samples to the total number of samples. A dataset is usually divided into training and testing sets to train a DNN model. The training set serves as the basis for model training, while the testing set is utilized to evaluate the model's performance (accuracy) against new and unseen data. This metric is utilized in our approach to determine the performance of a model and yields an accuracy result from 0% to 100%. The formula for accuracy is $acc = \frac{\text{True Predictions}}{\text{All Predictions}}$.

D. DNN IP Protection Attacks

Two common attacks on DNN IP protection methods are presented below: Fine-Tuning [42], [40] and Pruning [23]. The effectiveness of both attacks on our approach will be evaluated in Sect. VI.

Fine-Tuning. In Fine-Tuning [42], [40], the objective is to remove identifiers by continuing model training on a dataset that is comparable to the initial training dataset, assuming familiarity with the training process and hyperparameters to modify the model accordingly [42], [40]. Typically, in Fine-Tuning scenarios, a much smaller dataset than the one used for training is utilized with the goal of minimizing adjustments to the already learned features. Instead, only minor modifications based on the new dataset are desired to attain high accuracy. The learning rate (ratio how quickly a model adjusts its parameters to improve its performance) is commonly lowered in Fine-Tuning [40], [42] to preserve the learned features.

Pruning. Pruning [23] is a technique used to decrease the size of DNNs for deployment in resource-limited environments. By strategically removing model parameters, Pruning can be employed to remove an embedded identifier while maintaining acceptable model performance (cf. Sect. II-C) given that adversaries can arbitrarily modify parameters. Pruning entails selectively removing a predetermined percentage of parameters, referred to as Pruning level. Usually, in model Pruning, the focus is on removing the values with the lowest absolute values, as these parameters are deemed to have the least contribution on the overall performance of the model.

In the following, we will elaborate on the requirements and the threat model for a DNN IP protection method.

III. REQUIREMENT ANALYSIS

Motivation. We aim to develop a novel ownership verification method for ML models that addresses the limitations of existing approaches. The method must be robust against adversarial modifications, even if the verification process is publicly known. Furthermore, it should not rely on key secrecy or introduce additional parameters to be trained. For extended DNN training periods, intermediate model versions with decent prediction performance are stored as checkpoints. These checkpoints can be stolen and misused, so the novel technique should maintain the same identifier used for verification throughout the entire training process, ensuring that even model checkpoints are secure.

Considered Scenario. In our considered scenario, a data owner develops a proprietary ML model trained on a private dataset. The owner should be able to incorporate a unique non-secret identifier into the model, essentially serving as a distinct signature for ownership verification, which we call (non-secret) key or identifier interchangeably. After training, the model is deployed, either in a cloud environment or by selling it to others. However, the model is subsequently misused, e.g., stolen or illegally distributed and put into production by a third-party. It should be possible to identify the model by

3

analyzing the embedded key, regardless of any modifications, such as Fine-Tuning (cf. Sect. II-D), Pruning (cf. Sect. II-D), or key removal attempts, made by unauthorized third-parties. If the key remains unchanged, the owner can establish legitimate ownership and take action to claim their intellectual property.

Threat Model. The attacker has complete knowledge of the model and its architecture, including parameters. This degree of access is known as white-box access [39]. Furthermore, the adversary can modify the model parameters and architecture, an extremely powerful scenario that allows the attacker to change the model arbitrarily. This level of access also allows the adversary to partition the model at certain layers and use only parts of the model. In addition, similar attacks can be launched on the ownership verification method as those employed on watermarking approaches [21], [7], [55]. For instance, the attacker might modify the model by Fine-Tuning (cf. Sect. II-D) its parameters on a small subset of the training data. Furthermore, an adversary could fine-tune the model to fit a different dataset, e.g., with different output classes. Moreover, the attacker can launch Pruning (cf. Sect. II-D) attacks that entail eliminating specific connections among parameters within the model. However, the attacker does not have access to the original training data used to train the model, otherwise the attacker could train his own model instead. Furthermore, the attacker cannot tamper with the training and protection process, nor with the training data itself, because the training and protection are performed entirely on the model creator's side, without adversarial influence.

Adaptive Adversary. An adversary who is aware of the protection technique in place, may seek to manipulate or remove it. An adaptive adversary can use arbitrary techniques, leveraging the capabilities defined in the threat model, to remove the identifier. He could launch attacks specifically tailored to the protection method, adapting and extending known adaptive adversary scenarios from the Watermarking and Passporting domains. In general, the goal of the adaptive adversary would be to remove or replace the identifier, e.g., insert an arbitrary identifier. Removing the existing identifier would prevent the model creator from claiming ownership, while replacing it would give the adversary the ability to claim ownership. Specific attack scenarios are discussed in Sect. V and Sect. VI.

Objectives. Ideally, protected models should preserve their functionality and allow for unrestricted inference; however, the primary requirement is to maintain a transparent and robust identifier that can withstand attacks. Even as adversaries who have stolen the model adapt it for their own purposes, it is essential to preserve the identifier as evidence of copyright infringement in a legal context. By incorporating unique identifiers, the protection technique aims to secure the IP contained in models while preserving the functionality and performance of the model. Protection methods for DNNs must address the following six fundamental challenges.

1. Transparency. The ownership verification should be transparent, in particular, it should not rely on the secrecy of the used key. This entails the ability to perform multiple ownership claims.

2. Robustness. The protection technique should be resilient to adversarial attacks to prevent tampering or compromise. Specifically, the model's distinctive identifier must remain discernible, even after arbitrary adversarial attack attempts, such as Fine-Tuning [42], [40] or Pruning [23] as described above.

3. Fidelity. The protection should have negligible performance impact, e.g., accuracy should be upheld to levels of unprotected models.

4. Efficiency. The technique is expected to produce minimal overhead related to complexity and latency, e.g., significant training duration overhead should not occur for protected models.

5. Reliability. The protection process should ensure that unprotected models do not generate false positives, e.g., an unprotected model should not be identified as a specific protected one, while identifying protected models accurately.

6. Generalizability. The method should be versatile and adaptable to various datasets and model architectures. The flexibility ensures smooth integration into multiple ML architectures, making it valuable across various applications and scenarios.

In the following section we present our approach that solves the aforementioned challenges and fulfills the objectives.

IV. DNNShield Design

To protect DNNs from unauthorized access, we propose DNNShield, a method that facilitates reliable model identification after deployment by embedding unique identification layers that do not require secrecy and, thus, allow for repeated ownership verification of models. The method integrates non-secret and untrainable protection layers (referred to as locks) with fixed parameters (keys) well-distributed into the model architecture before training, without disrupting functionality or increasing training complexity. DNNShield is superior to watermarking schemes [47], [31], [43], [50], [51], [14], [34], [2], [54], [55], [32], [21], [27] because it does not require a secret key for ownership verification and, thus, allow for multiple verifications. The core novelty of DNNShield lies in the design of novel protection layers. Contrary to existing Passporting [20], [19] solutions, DNNShield has three advantages: The protection layers do not require training, which avoids overhead. Further, our approach is not limited to models that use convolutional layers and can also secure models that only use linear layers. Moreover, DNNShield can secure the model at any stage of the training process, including model checkpoints, with the same key. This is important as models become larger and their training time increases.

Each protection layer has a unique key (parameters) that is not changed during training. The key determines how the protection layer input, which is the output of the previous layer, is altered. For instance, the key defines how the data is scaled. Therefore, static operations on the output of the previous layer are applied and, hence, the behavior of subsequent layers is influenced. The altered data introduced by protection layers
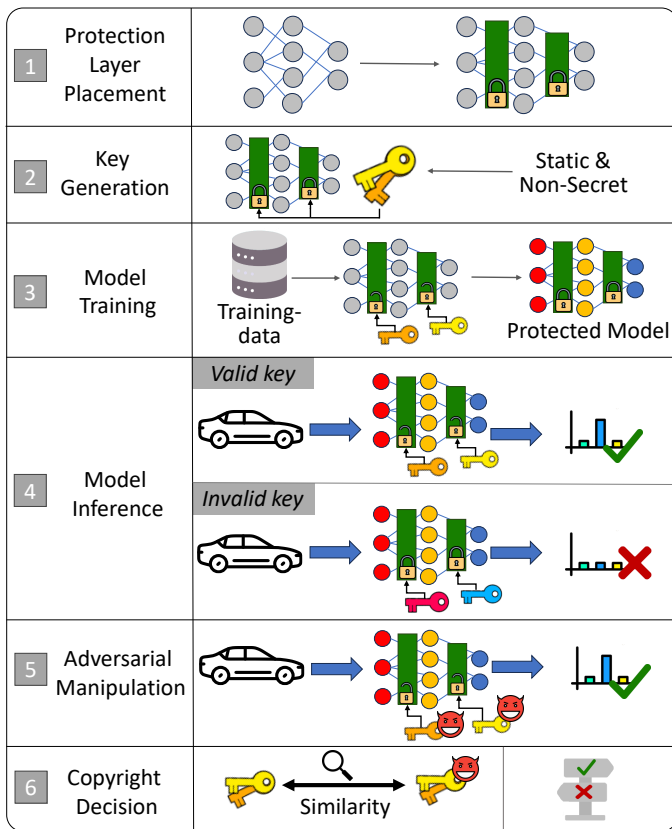
Fig. 1: Overview of the Approach.

ensures that the model parameters of subsequent layers are entangled with the specific protection layer key. Therefore, protection layers cause a significant model performance reduction if they are removed or their keys altered, rendering the model inoperable. In particular, the data processed by the manipulated model would deviate from the expected pattern, resulting in unexpected inputs to the layers following the protection layers. We name this characteristic of protection layers "protection property". Effectively, adversaries are compelled to maintain the integrity of the protection layers by using identical or very similar keys, facilitating ownership verification of legitimate owners by analyzing the used key. Similar to Passporting [19], [20] ownership verification schemes, the keys used by the suspected model are compared to the original keys used by the model owner, and if there is a high degree of similarity, model ownership can be claimed. Therefore, the unique keys of the protection layers can be published along with the model and then serve as a unique identifier, allowing for multiple ownership claims. This fulfills the Transparency requirement from Sect. III. To summarize, the protected model can be deployed and the lock and key's persistence allows the model to be identified. If adversaries attempt to remove the protection, the model performance is significantly degraded, rendering the model inoperable. In the following, we present the steps for protecting a model using DNNShield .

A. General Approach

Below, we present the life cycle of the DNNShield method consisting of six stages. All stages are depicted in Fig. 1.

Protection Layer Placement. In the initial stage, visualized as step 1 in Fig. 1, protection layers are incorporated at strategic points in the model architecture after unprotected layers, such as Convolutional or Fully-Connected (FC) layers (cf. Sect. II). We elaborate on the exact placement and amount of protection layers in Sect. IV-C.

The layers do not require training and, therefore, do not introduce additional training overhead. A protection layer modifies the output of the previous layer while maintaining the same output dimensionality, making it compatible with the following layer. This allows integration into existing model architectures at any point.[1] Therefore, the requirements Efficiency and Generalizability from Sect. III are addressed. The protection layer and its introduced data alteration become an integral component of the model since in DNNs the following layers rely on the output of the previous ones. Protection layers protect the parts of the model that follow them. Therefore, integrating multiple protection layers into a single model creates a more comprehensive defense by increasing the protection coverage.

Key Generation. Each protection layer is associated with a unique non-secret key in the form of static parameters that define how the data is altered. The protection layer serves as a gatekeeper, guaranteeing proper functionality only when the correct key is provided (referred to as protection property). Therefore, the second stage (step 2 in Fig. 1) involves defining a unique non-secret key for each of the previously defined protection layers. The composition, quantity, and length of the keys, which are the protection layer static parameters, vary depending on the protection layer type and the model architecture. In Sect. IV-B we will outline details of the key generation process. Once generated, the keys remain unchanged, including during model training and inference.

Model Training. Next, as shown in step 3 in Fig. 1, the model is trained with the protection layers and corresponding keys in place. The training process and the model creator's data is not manipulated. A model adjusts its parameters based on the specific data alterations from the protection layers defined by the used key. Thus, the parameters of the model are entangled with the keys. In the subsequent step, the trained and protected model along with the keys is then deployed, e.g., to the end-user or a web service.

Model Inference. Model inference (step 4 in Fig. 1) uses the keys defined in step 2 to unlock the protection layers. If the correct keys are provided to the protection layers within the model architecture, the data alteration will be identical to the one during training and the model will exhibit high performance. However, if the keys are not correct, e.g., manipulated,

---

[1]Otherwise, incorporating an extra dimension adapter layer would be required to adjust the output to the appropriate dimensions, which is not desired for easy integration and achieving minimal training overhead.

the performance will significantly decrease. This is caused by the protection layers not producing the expected data, causing subsequent layers that depend on the data produced by the protection layers to fail. In essence, a mismatch between the data from the protection layers and the learned parameters from the following layers occurs. Therefore, an adversary is compelled to use the keys defined in the second step; otherwise, the model becomes useless in terms of exhibited accuracy.

Adversarial Manipulation. In step 5 of Fig. 1, an adversary obtains and misuses the model and is capable of performing arbitrary manipulations as defined in the threat model in Sect. III. For instance, attempts to manipulate the utilized keys, the parameters of the model or its architecture could be performed (cf. Sect. III). The adversary's goal is to unlock the protection layers with different self-defined keys or remove them while maintaining the model's accuracy. This allows for unrestricted use of the model as copyright claims can no longer be made.

Verification. In step 6 of Fig. 1, all keys of a suspected model are analyzed. A comparison is drawn between the original keys from all protection layers and those from the suspected model. Our approach employs metrics that are suitable for the novel version of protection layers. Copyright infringement will be evident if significant similarity is detected. We will elaborate on how to reliably measure the similarity of different keys in Sect. IV-B, by utilizing the cosine similarity and a newly introduced metric.

B. Protection Layer Instantiations

In this section, we present two specific instantiations of protection layers. One is based on element-wise multiplication called "Hadamard layer", while another one relies on shifting the order of outputs called "Permutation layer".

Hadamard Layer. Those layers perform an element-wise multiplication (cf. Sect. II-B) between two matrices of equal dimensions. The protection layer can be used in conjunction with any convolutional or FC layer (addressing the Generalizability requirement of Sect. III). As shown in Fig. 2, the protection property is implemented by using the output values of the preceding layer as input for the protection layer and multiplying each of them with the corresponding key value, visualized by the green boxes and the keys in the center. Thus, the key must have equal dimensionality as the output (usually c, w, h for convolutional layers or w, h for FC layers as explained in Sect. II-B) of the preceding layer. In essence, every output value of the preceding layer, undergoes scaling by a different key-defined constant factor. The following layers adjust their functionality to the particular scaling introduced by the Hadamard layer during model training.

Robustness. If the scaling defined by the key deviates significantly during inference, it will result in changes to the output values. As all subsequent layers rely on values produced by the Hadamard layer during training changing the keys will lead to decreased performance. This fulfills the Robustness
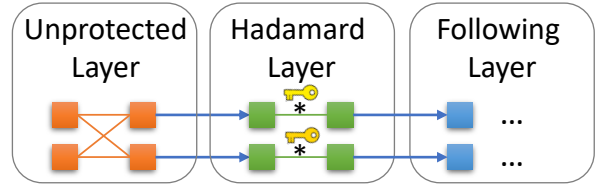

Fig. 2: The Hadamard layer Building Block.

requirement from Sect. III, as the layers cannot handle values that deviate from the known scale. A misalignment in value scaling impacts the entire model, including all subsequent layers, as each layer depends on the output values of the previous layer. We show the functionality and robustness of the Hadamard layer through an empiric evaluation in Sect. VI.

Key Generation. For every output value in the preceding layer, an element-wise multiplication is executed with a specific value provided by the non-trainable key. As such, the key has to contain one scaling factor for each output value. While there are no general limits on the values within the key, it is important to choose value ranges carefully to prevent gradients from exploding or vanishing [4]. Values that approach zero create particularly small output values and can subsequently lead to vanishing gradients. Similarly, excessively large key values may lead to exploding gradients, and should therefore be avoided as specified by the Fidelity requirement of Sect. III. We propose to use randomly selected values that are uniformly distributed within a range of -1.0 to 1.0. Negative and positive values are included in the key to enable changing signs, which, we believe, makes it more difficult to remove protection layers due to the increased entanglement with model parameters. We explore the effects of different key ranges in Sect. VI.

Copyright Verification. The keys of the Hadamard layers serve as model identifiers. Therefore, the keys of a suspected model are compared one by one to the original keys. For copyright verification, the keys are represented as flattened vectors and we suggest using cosine similarity (cf. Sect. II-C) for comparison. While the specific comparison metric may differ implementation-specific utilizing cosine similarity can provide a reliable measure of similarity, essentially fulfilling the Reliability requirement (cf. Sect. III). If there is a high similarity between the original and suspected model keys, it is likely that the entire model or parts of it were copied from the original model, providing evidence of copyright infringement. Similarity calculations can be performed for each Hadamard protection layer key individually, providing a detailed understanding of which model layers were stolen. We establish an insensitive guideline regarding the cosine similarity for ownership verification through an empirical study in Sect. VI. Furthermore, we empirically show that, despite third-party manipulation post-deployment, the following layer's dependence on a specific value range cannot be eliminated.

Next, we introduce the Permutation layer, a protection layer that was specially crafted to be used with convolutional layers (addressing the Generalizability requirement from Sect. III). The robustness of the Permutation layer relies on its usage in

conjunction with convolutional layers. It could function as an alternative or extension to the Hadamard protection layer.

Permutation Layer The Permutation layer shifts the order of input data, making it an effective mechanism for implementing a protection layer that adheres to the protection property. The intuition behind the layer is to allow the convolution process begin at a unique non-standard starting position, rather than the top-left corner, while still processing the entire image sequentially. The key defines the specific starting position for each kernel. The resulting output retains the dimensions from the unprotected layer and preserves local relationships between adjacent output values while adopting a rearranged order. Recall, that each kernel in the convolutional layer (cf. Sect. II-B) has its own output channel. Instead of modifying the starting positions of the kernels, one can achieve the same effect by shifting the outputs of the convolutional layer on a channel-wise basis. An example of a shift from a single channel to the right within a 3x3 matrix is shown in Fig. 3. The left matrix indicates the original order, while the middle matrix shows a shift to the right by one position and the right matrix shows a shift by eight positions to the right. The non-trainable and static key must have the same length as the number of output channels (cf. Sect. II-B) of the preceding convolutional layer to ensure proper operation. During training, subsequent layers will adjust their parameters based on the output order introduced by the Permutation layer.

Robustness A deviation from the altered order (defined by the key) of the Permutation layer disrupts the learned features, rendering the following layers incapable of accurate operation, fulfilling the Robustness requirement of Sect. III and adhering to the protection property. Random shuffling cannot be performed because convolutional kernels may partially overlap during the convolution operation, effectively operating on the same pixels multiple times. Consequently, the outputs exhibit a sensitivity to their order, meaning randomly shuffling them can significantly impair the model performance as spatial information is lost. To maintain the correct relative positions, the values are shifted (addressing the Efficiency requirement from Sect. III). We confirm the protection effectiveness of this layer through empirical evaluation in Sect. VI.

Key Generation Recall, that the number of output channels is dependent on the number of kernels utilized in the convolutional layer (cf. Sect. II-B). In the Permutation layer, all outputs from a channel are shifted to the right by a certain key-defined factor. For instance, the values in the first channel will be shifted by three positions, while values from the second channel will be shifted by five positions. Thus, the key consists of a single integer value for each channel of the preceding convolutional layer. The key values are randomly generated between one and the number of output values minus one in the respective channel. This indicates a shift to the right by one position or a shift to the left by one position. The careful selection ensures that the layer shifts at least one position to the right and in each case alters the input's order.

Copyright Verification. Instead of directly comparing the keys

Fig. 3: A shift to the right by 1 and 8 positions is shown.

of the Permutation layers as done with Hadamard layers, we examine the outputs of Permutation layers to establish ownership verification. This is done to mitigate adaptive adversary attacks, which will be explained in Sect. V. To achieve this, we first create a synthetic input sample where each value is unique, such as a matrix with entries ranging from 0 to the number of data points as shown in Fig. 3. Then, we analyze the outputs of the input sample generated by two Permutation layers. The first Permutation layer is used with the originally employed key, while the second uses the key found in the suspected model. Given the absence of suitable methods to assess the similarity between two shifted outputs, we devised a novel metric. This metric involves calculating the channel-wise number of shifts required to move one of the outputs to the left or right, maximizing the resemblance to the other. Similar to brute-forcing, every possible shift is performed and the output that produces the highest similarity is selected. In particular, the cosine similarity is utilized to measure the similarity between the two outputs.

We start by defining $k = \frac{\text{PossibleShifts}}{2}$ as the maximum number of shifts (divided by two as shifting can be done to the left or right side). Next, we define the Permutation Accuracy (PAC) as $\text{PAC} = 1 - r/k$. Here, r specifies the minimum count of shifts that maximized the similarity between both outputs (considering both left and right direction). This metric defines the percentage of similarity between the output values of two Permutation layers in terms of value order. As the output order is defined by the key, essentially, the key's similarities are measured. Thus, ownership can be claimed based on the similarity of the PAC metric.

C. Protection Layer Placement

In the following, we elaborate on the placement and number of protection layers in model architectures. Protection layers can be placed arbitrarily, as long as they are preceded by a convolutional or FC layer. Integrating multiple protection layers distributed at different locations provides more comprehensive protection, as each layer protects the parts of the model that follow it.

DNN Models are often already split into architectural parts as their architecture repeats itself, e.g. [24], [30], [26], [25]. For instance, the ResNet [24] models are constructed from a number of basic blocks. Similarly, Transformer-based models consist of a number of Transformer blocks [46], [5]. The amount and exact placement of Protection layers is an insensitive parameter as we show in Sect. VI-A. Nevertheless,
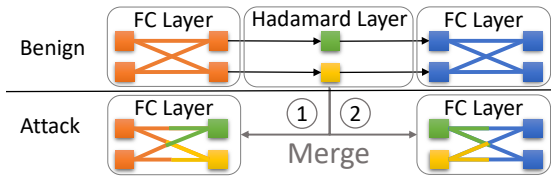
Fig. 4: Merge of Hadamard layer into Fully-Connected (FC) layers.

Fig. 5: Kernel pattern modification attack.

we propose placing one protection layer inside each model block after the first FC or Convolutional layer, providing comprehensive model protection. The last layer of a model is the output layer, here no Protection layer is added as no layer uses these outputs for further computation.

For instance, a ResNet-18 is divided into its nine Basic Blocks where each block has roughly the same number of layers as shown in Appendix Table VI. The first convolutional layer of each block is preceded by a Protection Layer. Likewise, a Transformer-based model is protected by placing one Protection layer after each FC layer from each Transformer block. Smaller models can also intuitively be split into model parts where each part has roughly the same amount of layers as demonstrated in Appendix Table IV and Table V.

V. SECURITY ANALYSIS

In the following, we discuss attacks that merge the protection layer with neighboring layers and attacks that manipulate convolutional kernel patterns to imitate the Permutation layer. Additionally, the feasibility of partitioning the protection layer and the robustness of the PAC similarity metric (cf. Sect. IV-B) are discussed.

Merge. If a Hadamard layer is used in conjunction with a Fully-Connected (FC) layer, an adversary could merge the keys of the Hadamard layer with a FC layer (merge attack), as illustrated in Fig. 4. The key of the Hadamard layer can either be multiplied with the parameters of the preceding layer (annotated with 1) or with the following FC layer (annotated with 2). Merging one of the FC layers and the Hadamard layer would allow for the removal of the Hadamard layer. We elaborate on the mathematical details of the multiplication in Sect. A. However, this attack still leaves identifiable traces that can be used for copyright claims.

Assume that an adversary performs the merge attack and, therefore, obtains a manipulated (FC) layer that does not require the Hadamard layer for proper outputs anymore, as visualized as the bottom left FC layer or the bottom right FC layer in Fig. 4. The manipulated layer comprises of parameters that are multiplied by the key of the Hadamard layer. For example, in the lower left corner, the outgoing paths from the first orange neuron are multiplied by the green factor of the Hadamard key. The key of the Hadamard layer can still be extracted from the manipulated layer by reversing the merge attack. The key is obtained by dividing the parameters of the manipulated layer by the parameters of the original FC layer. As visualized in Fig. 4, each neuron has multiple paths and
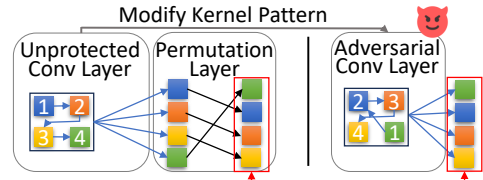
for a simple merge attack each parameter would yield the same key factor. However, in case the adversary manipulates a parameter, all the paths contributing to one output must be considered. Hence, we average all multiplication factors determined by all paths contributing to one output. The division yields a multiplication factor corresponding to the key for each path. The mathematical details of the reversal process are outlined in Sect. B. After averaging, the obtained values have identical dimensionality as the originally used key and can be easily compared using the cosine similarity metric. Furthermore, we demonstrate through a robustness evaluation in Sect. VI-D that this attack combined with Fine-Tuning the resulting manipulated model is also ineffective.

Convolutional layers cannot incorporate the Hadamard layer's key into the parameters, as convolutions share the same parameters for multiple outputs. Therefore, the merge attack would require multiplying a single parameter with different factors simultaneously, which is not possible. We elaborate on the precise details in Sect. C. Thus, the merge attack is infeasible for both FC and convolutional layers.

Convolutional Pattern Modification. Permutation layers shift output values by a certain factor (cf. Sect. IV-B). Rather than modifying the order from the convolutional layer, the same rearrangement could be produced by manipulating the starting position of the convolutional kernel as shown in Fig. 5. The figure visualizes, that modifying the order in which the convolution is performed on the right-hand side results in the same output order as introduced by the Permutation layer in the center. This attack allows for removal of the Permutation layer. However, ownership can still be proven. This is done by calculating the PAC metric on the output of the adversarial convolutional layer and the output of the original convolutional layer combined with the Permutation layer. As only the order of output values is important, the parameters of the manipulated and original convolution layers are unified, e.g., they have the same parameters. Thus, the order in which the output is rearranged is analyzed for both layers, and ownership can be claimed as the same order is produced, rendering the attack infeasible.

Protection Layer Split. A potential attacker may try to bypass the protection mechanism by dividing the protection layers into multiple layers, each performing a part of the overall function. However, the cumulative effect of these layers would replicate the original protection layer. Therefore, it can be identified by examining the architecture, as the layers would need to be sequentially connected. Consequently, the separated layers can be easily reassembled into a single layer, and
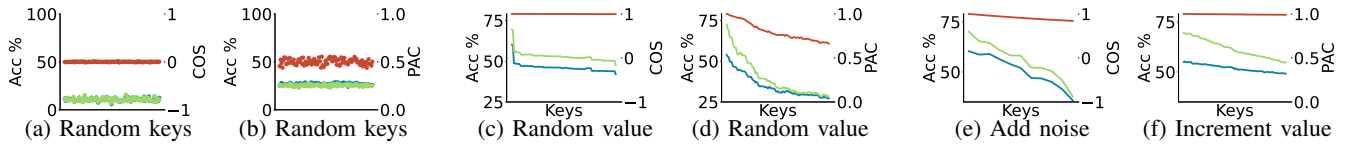
Fig. 6: CNN robustness protected by Hadamard(a,c,e)/Permutation layers(b,d,f) to key manipulation, e.g. random key, replacing of key value, and, adding noise to key. Red line depicts key similarity, Green and Blue depict train and test accuracy.

ownership can be claimed.

Similarity Metric Resilience. The proposed PAC metric relies on a minimal distance to assess the similarity between Permutation layer outputs. This prevents an adversary from subverting the PAC similarity metric by introducing a single value that has not been shifted but rather randomly permuted. Even if the injected value disrupts the alignment of the two outputs, making it infeasible to align them with a shifting operation alone, the PAC metric can still determine the number of shifts required to maximize their resemblance. As a result, this renders the attack ineffective.

VI. EVALUATION

Model Architectures and Datasets. Our approach was assessed on various model architectures with differing sizes commonly utilized in image classification domains, including a Fully Connected network (FCN), a Convolutional network (CNN), and the ResNet-18 [24] architecture. Further information about these models can be found in Sect. D. We chose the Vision domain to showcase the effectiveness and robustness of DNNShield , aligning with prior research ([47], [31], [43], [50], [14], [34], [2], [54], [32], [21]), leveraging commonly used datasets that focus on image classification namely MNIST [15], CIFAR-10 [29], and GTSRB [41]. Throughout the experiments, we varied the model architecture, and training dataset systematically to showcase the Generalizability (cf. Sect. III) of our approach. The models underwent ten epochs of training utilizing the Adam optimizer with a learning rate of *0.001*, unless explicitly stated otherwise. In all experiments, the mean similarity of all keys is reported. The experiments were conducted using PyTorch, a leading Python-based machine learning library [44], [37], [48], on a server equipped with 96 processing units, 128GB main memory, and an AMD EPYC 7413 24-Core Processor (64-bit). We accessed an NVIDIA A16 GPU via CUDA [36], which has four virtual GPUs, each with 16GB of GDDR6 memory.

A. DNNShield 's Functionality

To showcase the general functionality of DNNShield , we employ Hadamard and Permutation protection layers to protect the CNN model (cf. Sect. D). The comparison results with an unprotected model, as well as the best key ranges, will be evaluated in Sect. VI-B. After training on the CIFAR-10 dataset [29], training accuracies of *69.89%* for the version with Hadamard layers (H-Model) and *75.77%* for the version with Permutation layers (P-Model) were achieved, showing that working models can be trained with protection layers in place. The number of values in the key, e.g., the key size of the

H-Model was around *3.2%* compared to all model parameters and around *0.01%* for the P-Model. First, the functionality of the protection layers is evaluated with three experiments. The goal is to ensure that the protection layers reduce the model accuracy for keys that deviate from the reference keys (as defined by the Robustness requirement from Sect. III) which validates our claims that the protection layers adhere to the protection property (cf. Sect. IV). These experiments showcase that the keys are entangled with the model parameters and, thus, have an impact on the model performance in case of manipulation. In the first experiment, we establish a baseline by replacing the key with randomly generated false keys and showing which key similarity and model performance is achieved. Next, we show that small deviations in the key's values have an impact on the model accuracy. Further, we show the impact of adding different levels of noise to the key of a protection layer.

Position and Amount of Protection Layers To assess the impact of the number and placement of protection layers within a model, all combinations of one, two, three, and four protection layers integrated into the CNN model (cf. Appendix Table V) were evaluated. The results, depicted in Table I, show that neither the number of protection layers nor their specific positions within the model seem to affect the performance. In particular, the columns of the table show the different combinations of active protection layers, e.g. Protection Layer 1 to 4. The rows show the mean model performance and its variance during 20 training epochs. It is evident, that the performance does not change significantly for any Protection Layer combination. This confirms, that the placement and quantity of protection layers is an insensitive parameter. To further showcase the position independence, we evaluated placing the Protection layers after the first within each Basic block of the ResNet-18 [24] as shown in Table VI. We also place the Protection layer after the second Convolutional layer within each block. Varying the position of the Protection layers in the ResNet-18 [24] architecture did not alter the outcome of performance or robustness experiments.

Model Refinement In scenarios involving data concept drift, where fine-tuning the model is required at later stages to adapt it to new data [22], it is crucial that the model can be refined after it was protected without requiring re-training from scratch. To evaluate the ability to fine-tune a protected model, we conducted an experiment, training a ResNet-18 model on the GTSRB dataset for 10 epochs, achieving a training accuracy of 86.95%. The last layer of the model was then replaced to adapt it to the CIFAR-10 dataset. Afterward,

9

Combination	{1}	{2}	{3}	{4}	{1,2}	{1,3}	{1,4}	{2,3}	{2,4}	{3,4}	{1,2,3}	{1,2,4}	{1,3,4}	{2,3,4}	{1,2,3,4}
Mean Accuracy	98.4	97.9	98.9	98.9	97.7	98.6	98.4	97.7	97.9	98.8	97.9	97.7	98.4	97.8	97.6
Accuracy Variance	0.9	1.1	0.9	0.9	1.5	1.1	1.0	2.1	1.1	1.0	1.9	2.3	1.4	1.7	3.0

TABLE I: Different Combination of protection layer P1, P2, P3, and, P4 for the CNN model trained on MNIST for 20 epochs.

the model was fine-tuned on the CIFAR-10 dataset for an additional ten epochs starting at an accuracy of 71.59% and ending with 93.5%. The used keys remained identical throughout the whole process. These results demonstrate the ease of refining existing models on new datasets, as long as the legitimate keys are preserved. Thus, DNNShield enables model refinement after protection, eliminating the need for complete retraining, resulting in significant resource savings.

Key Replacement. To determine the functionality and validate the protection property of the layers, we evaluated the trained and protected models in a setup where the original key was replaced with 100 different randomly generated false keys. [2] We expect the key similarity to be low, however, also the model performance should significantly drop as the wrong key is used. In Fig. 6a the results are shown, we omit the x-axis label due to space reasons. The green dots represent the accuracy on the train set, while the blue dots represent the accuracy on the test set. Note, that the blue dots are barely visible as they overlap with the results from the train set. The red dots represent the cosine similarity of the randomly generated key compared to the original key. The cosine similarity remains consistently low, as it is at approximately 0 for all randomly generated keys, indicating that no false positives occurred as desired by the Reliability requirement from Sect. III. As expected, we can see, that the H-Model exhibits poor performance, akin to that of a naïve classifier (around 10%) due to the usage of false keys, which is represented by the green and blue dots at the bottom of Fig. 6a. The drop in accuracy from around 70% to 10% partly fulfills the Robustness requirement from Sect. III. Meanwhile, for the P-Model, a comparable outcome is yielded as depicted in Fig. 6b. Again, the green and blue dots represent the training set and test set accuracies, while the red dots represent the PAC (cf. Sect. IV-B) metric. The model exhibits poor accuracy of around 30% (compared to 70% for the original key), while the PAC (cf. Sect. IV-B) remains at 0.5. The PAC value can be explained by the fact that a large sample of numbers from a randomly and uniformly distributed population is taken. Therefore, the average will tend to be close to the center of the distribution. We conclude that the PAC value must be above 0.5 for reasonable ownership claims. In summary, we have shown, that replacing the key with random keys yields poor model performance as well as low key similarity. Showing that both protection layers adhere to the protection property (cf. Sect. IV).

Incremental Key Value Replacement. An attacker may replace a single value instead of the entire key, with the goal of decreasing key similarity while maintaining high performance. Therefore, we show that by iteratively replacing key values with random values, either the model performance degrades or the key's similarity remains high. The generated key values were again in a valid range. Thus, we performed an evaluation that involved iteratively replacing zero to 100 randomly chosen values within the original key with randomly generated values, e.g., the first iteration did not change any values and each subsequent iteration modified another random value. We limited the evaluation to 100 generated keys since the trend was already apparent. The results are depicted in Fig. 6c, here the red line indicates the cosine similarity compared to the reference key. The green and blue lines represent the training and test set accuracies of the H-Model. It is visible that the accuracy drops from 69% with the original key to around 56% with just three values replaced. For the test set an accuracy drop of the same magnitude can be observed. The key similarity remains high above 0.99. The P-Model exhibits a comparable outcome, as illustrated in Fig. 6d. Again, the red line represents the key's PAC, while the green and blue lines denote the train and test set accuracies. The accuracy declines from over 75% to around 27% for both the training and testing sets. Still, a PAC of more than 0.66 is yielded. For a PAC of 0.85 the model accuracy dropped to below 43%. Both experiments indicate that the keys from the protection layers are sensitive to small changes, as they result in a significantly reduced model performance, indicating that the keys are entangled with the model parameters.

Add Noise to Key. Another method to tamper with the key is to add noise to the key's values. Thus, we add different levels of random noise in an iterative process to the keys of the protection layers. In particular, for Hadamard layers, we add randomly generated noise ten times in an interval of -0.2 to 0.2 to the key. For Permutation layers, 100 times a randomly selected key value is increased by one. The results for the H-model are shown in Fig. 6e. The meaning of the colors in the figure is identical to the experiments for the H-model. The leftmost point in Fig. 6e shows the result where no noise was added. In subsequent experiments, the added noise was further increased. In total, the random noise was added ten times to the H-Model, which prompted a decrease in accuracy by about 32% to a value below 37%, while the cosine similarity of the keys remained high at more than 0.84. The results for the P-Model are shown in Fig. 6f (the meaning of the colors is identical to previous P-Model experiments). Further incrementing random key values up to 100 times prompts a drop in accuracy from around 69% to 54% for the training set and 54% to 49% for the test set. The PAC remains at above 0.98. Both models appear to exhibit good resilience against

---

[2] The keys were generated in a valid range, e.g., -1.0 to 1.0 for Hadamard layers and 1 to the number of channels minus one for Permutation layers.

Model	Unprotected	Hadamard	Permutation
CNN	99.826%	98.795%	**99.861%**
ResNet-18 [24]	**98.894%**	97.788%	97.756%

TABLE II: Accuracy results for CNN trained on MNIST [15] dataset and ResNet-18 [24] trained on CIFAR-10 [29].

adding noise to the key values, as an accuracy drop is clearly visible.

In summary, the experiments indicate a strong correlation between the key similarity and model performance. A dissimilarity between modified and original keys leads to a drop in model accuracy. Furthermore, the key metrics only exhibit high similarity in case very similar keys are utilized, fulfilling the Reliability requirement from Sect. III. We conclude that our approach is effective and reliable as the models achieve high accuracies when the correct keys are utilized. However, the model performance significantly declines and the key similarity is poor if the keys significantly deviate from reference keys, fulfilling the Robustness requirement from Sect. III. Based on our experiments, we suggest that a similarity value greater than *0.8* for both metrics could be used as a guideline for ownership verification. This value provides a good balance between robustness and reliability. However, the threshold is an insensitive parameter.

B. Fidelity and Efficiency

In the following we first determine the best key range for the Hadamard protection layer, exhibiting the best performance and robustness. Afterward, we present evaluation results for measuring the overhead induced by the protection method.

Key Range. We confirm the key parameters for the Hadamard protection layer, by evaluating the potential influence of exploding and vanishing gradients [4] associated with high and low values (cf. Sect. IV). Additionally, we assessed the extent to which the inclusion of positive and negative values impacted the performance of the protection layer. All experiments were conducted with a ResNet-18 [24] model on the CIFAR-10 dataset [29].

Our experiments show that Hadamard Protection layers with high key values (generated randomly between -10.0 to 10.0) do not impact the protective capabilities. Nevertheless, they resulted in a *2.57%* decrease in accuracy compared to regular keys ranging from -1.0 to 1.0, and therefore, should be avoided. Similarly, low key values (generated randomly between -0.1 to 0.1) also resulted in a decrease of accuracy by *2.11%*. The model performance is increased by *7.38%*, using only positive values ranging from 0.0 to 1.0. However, when replacing the original key with randomly generated ones, the model protected with the key that did not include negative values (range from 0.0 to 1.0) exhibited about *5%* higher accuracy when used with random keys. Thus, the robustness, e.g., the drop in accuracy when utilized with false keys is positively impacted by including negative values. Therefore, we argue that inclusion of negative values is beneficial due to increased resilience. We conclude that adding negative values

enhances the entanglement between the key and model parameters, resulting in increased robustness. Thus, all experiments are conducted with keys ranging from -1.0 to 1.0.

Fidelity. To measure the accuracy impact, we first trained two unprotected baseline models: A ResNet-18 [24] on CIFAR-10 [29] trained for 30 epochs and a CNN model trained on MNIST [15] for 30 epochs. The training duration was extended to demonstrate that the models are fully optimized, meaning a high level of accuracy has been achieved with little potential for further improvement. Next, we included Hadamard and in the second run Permutation layers into the models and again trained in the same manner. The results for the training accuracies are visualized in Appendix Fig. 11. The figure indicates a minimal difference between the models. In the following, we report the evaluation of the whole dataset (train and test combined) reported in Table II. After 30 training epochs the unprotected ResNet-18 [24] model achieved an accuracy of *98.89%* while the configuration with Hadamard and Permutation layers achieved accuracies of *97.79%* and *97.76%*. Accordingly, we have observed that both protection layers cause a performance drop of around *1%*, which we consider negligible as this can also stem from training randomness. Using another random seed leads the unprotected ResNet-18 [24] model to achieve an accuracy of *97.20%* which shows that performance fluctuations occur. Similarly, the overhead introduced by the Hadamard Protection layers in the CNN model is around *1%* and the version with Permutation layers performs *0.035%* better than the baseline. The impact of protection layers is more pronounced during the initial stages of training. However, their influence diminishes as training progresses, and ultimately, models with and without protection layers achieve almost identical accuracies. Both models, in all three configurations, display a similar training accuracy curve and accuracy values, leading us to believe that the impact of our approach on the model performance is negligible fulfilling the Efficiency requirement from Sect. III.

C. Generalizability

To demonstrate the generalizability, we protect a ResNet-18 [24] model using Hadamard layers (H-ResNet) followed by another version utilizing Permutation layers (P-ResNet). We trained both model variations on the CIFAR-10 dataset [29] and the GTSRB dataset [41]. The H-ResNet achieved a training set accuracy of *80.48%* on CIFAR-10 [29] and an accuracy of *94.89%* on GTSRB [41]. Furthermore, the P-ResNet achieved an accuracy of *82.06%* on CIFAR-10 [29] and *97.18%* on GTSRB [41]. To ascertain the robustness for the H-ResNet and P-ResNet trained on CIFAR-10 [29], we again conducted the same experiments as in Sect. VI-A. In particular, three experiments were conducted. The first experiment replaced the original key with 100 randomly generated keys. The second experiment replaced a single key value, and the third experiment added noise to the key. The results are depicted in Appendix Fig. 10 and show that the ResNet-18 [24] model behaves very similar to the previously assessed CNN model. Therefore, we conclude that both

		All Params		Key Params	
		Same LR	1/10 of LR	Same LR	1/10 of LR
(1)	**COS**	**0.9975**	**0.9999**	**0.9855**	**0.9994**
(2)	Test Set Acc	97.14%	93.28%	76.03%	64.63%
(3)	Δ Train Acc	-13.27%	-1.74%	-2.63%	-1.64%

TABLE III: Fine-Tuning accuracy (Acc) for ResNet-18 [24] protected using Hadamard layers. The table depicts results for all parameters and only key parameters being tuned, both with the same learning rate (LR) and 1/10 of the LR.

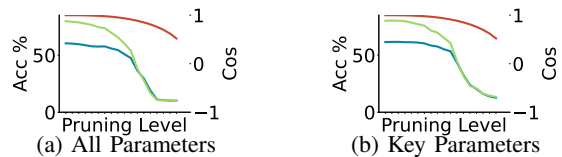(a) All Parameters   (b) Key Parameters

Fig. 7: Pruning from 5% to 90% for the H-ResNet trained on CIFAR-10 [29]. The Red line depicts the cosine similarity, while the Green/Blue lines depict train/test accuracy.

protection layers are applicable to multiple different datasets and model architectures.

Applicability to Transformer-Based models To demonstrate the applicability of DNNShield to Transformer-based models, e.g. Large Language Models [49], [46], [5], we protect a Vision Transformer [17]. Transformer models are naturally divided into processing blocks. Our model consists of 7 Transformer blocks. The embedding size and the Fully-Connected (FC) layers input dimension of the Transformer blocks are 384. Further, the model utilizes 12 Self-Attention modules. We protect the model by inserting a Hadamard Protection layer after the first FC Layer within each Transformer block. The unprotected Vision Transformer achieved an accuracy of 83.45% with protection layers in place. Similar to the ResNet-18 [24] experiments, we evaluated the robustness of DNNShield with three experiments. The first experiment was repeated 100 times and evaluated the model's performance using randomly generated keys, mimicking an adversary trying to utilize the model with a different key. Each time, a significant performance drop of around 70% occurred. Incrementally replacing single key values (up to 100) resulted in a negligible performance drop (less than 1%), however, the cosine similarity of the key remained above 99.94%. Finally, adding noise to the key until the cosine similarity dropped to 80% caused a performance drop to around 20.59% (a decrease of approximately 62%). These results are very similar to the other models exhibiting the same behaviour. Thus, DNNShield can be effectively applied to Transformer-based architectures as well.

D. Robustness and Reliability

In this section we evaluate the robustness of our approach against more sophisticated adversarial modifications. Common attacks, such as Fine-Tuning (cf. Sect. II-D) and Pruning (cf. Sect. II-D), can only be applied for the Hadamard Protection layer since the Permutation Protection layer cannot be optimized using gradient descent techniques. This is due to the Permutation Protection layer simply rearranging the order of outputs without modifying their values.

However, the Hadamard layer introduces key values (parameters) that adversaries can directly optimize using gradient descent, making it applicable to common attacks such as Pruning and Fine-Tuning. Therefore, we evaluated the vulnerability of the Hadamard Protection layer to these two scenarios.

Fine-Tuning. In fine-tuning attacks (cf. Sect. III), attackers alter the parameters of trained models to circumvent security

measures while preserving the accuracy. DNNShield is not vulnerable to such attacks because its security is dependent on the key layers that remain unchanged even if other model parameters are manipulated. We considered two scenarios how malicious actors may adapt Fine-Tuning to DNNShield . In the first, adversaries treat the values in the key layer as if they belong to the model parameters, e.g., make them trainable. The goal is to compromise the protection mechanism by Fine-Tuning all model parameters and key values. Second, adversaries freeze all model parameters except for the key values, which are then fine-tuned in order to alter them while preserving the accuracy. The third scenario, training only the model parameters without the key values, is not beneficial, as in this case, the model identifier would remain unchanged.

To evaluate DNNShield 's robustness against adapted Fine-Tuning scenarios, we continued training the H-ResNet on the CIFAR-10 [29] test set. We trained for another ten epochs with the same learning rate, as well as with 1/10 of the original learning rate (similarly to [47], [9], [52], [34], [2], [31]). The evaluation results are depicted in Table III. In all tested scenarios the cosine similarity between the modified key and the original key was above *0.98* as can be seen in line 1 of Table III. Furthermore, line 2 reports the accuracies of the test set which was used for Fine-Tuning. Line 3 of Table III shows the change in the training set accuracy. Therefore, we argue that the approach is robust against Fine-Tuning a protected model. To further evaluate the robustness of our approach we fine-tune a ResNet-18 [24] model, protected using Hadamard layers, and trained on GTSRB [41] for 10 epochs on the CIFAR-10 dataset [29] for another 10 epochs. This process involved replacing the last layer of the model as the number of output classes changed, therefore, we only evaluated the scenario where all model parameters including those of the key layer are fine-tuned. The learning rate was set to 1/10 of the original learning rate as for higher learning rates the accuracy results degraded. After the Fine-Tuning, the cosine similarity of the key was above *0.99* while the train and test accuracies of the GTSRB [41] dataset were *86.78%* and *94.18%* respectively. Our results show that an adversary can not remove the identifier by adapting the model to his purposes. The approach exhibits strong resilience against adaptive adversaries performing Fine-Tuning attacks, essentially fulfilling the Robustness requirement from Sect. III.

Pruning. Besides Fine-Tuning, we investigate another common attack called Pruning (cf. Sect. III) similar to [47], [14], [9], [52], [34], [31]. We again employ the two scenarios from

Fine-Tuning, as Pruning only the model parameters would leave the key layers values untouched. The results are depicted in Fig. 7. The blue line depicts the test set accuracy, while the green line represents the training set accuracy. The red line depicts the cosine similarity of the key. First, we prune all model parameters including those of the key layers and second, we only prune the values from the key layers. We used the ResNet-18 [24] model which was trained on CIFAR-10 [29] for ten epochs and progressively pruned the model parameters and the key layers. We started with Pruning *5%* of the lowest values and increased the Pruning value by *5%* until we reached *90%*. As shown in Fig. 7a, in case all parameters are pruned, for a key similarity of *0.85* (reached at a Pruning level of *65%*) the accuracies are at *27.76%* for the test set and *29.5%* for the train set. Similarly, as shown in Fig. 7b, in case only the key parameters are pruned for a key similarity of *0.85* (also reached at a Pruning level of *65%*) the accuracies are at *31.73%* for the train set and *32.57%* for the test set. We conclude, that in both scenarios the model performance is correlated with the similarity of the key. Hence, the approach is resilient against Pruning attacks, adhering to the Robustness requirement from Sect. III.

Adaptive Adversary. An adaptive adversary could integrate the protection layer's parameters into the preceding FC layer (see FCN model in Sect. D). This attack strategy was discussed in Sect. V and illustrated in Fig. 4, where we determined that the key could be extracted from the merged parameters using the process outlined in Sect. B. A FCN model was trained on the MNIST dataset [15]. Next, both Hadamard Protection layers from the FCN model were multiplied into the parameters of the preceding FC layers. Subsequently, the model underwent Fine-Tuning on the test dataset using the same learning rate used in the regular training phase. Our evaluation results show that even after ten Fine-Tuning iterations, the key's values could be extracted from the model parameters with a cosine similarity exceeding *0.99*. Therefore, we conclude that multiplying the Hadamard key values into parameters of a FC layer is not an effective means of circumventing the protection mechanism. To ensure that unprotected model do not yield high key similarity in case the previously described extraction process is performed, we conducted a second experiment regarding retrieval of the key from an unprotected and trained model. Here, the cosine similarity was very low at *-0.13%*, therefore, we consider the Reliability requirement from Sect. III fulfilled.

To further assess the effectiveness of replacing protection layers with FC layers, we conducted an additional experiment. While replacing Permutation layers is ineffective, as ideally, they would simply learn how to rearrange the output values in a manner already accomplished by the Permutation layer, replacing Hadamard protection layers with FC layers holds limited promise. The idea was introduced in [11]. It is noteworthy that replacing each protection layer in the ResNet-18 [24] model with a FC layer introduces a parameter overhead of 28 times, e.g., the amount of parameters is increased from 11M
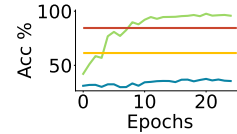

Fig. 8: Model test accuracy (green line) where key layers were substituted with FC layers. The blue line depicts the training accuracy, the Red/Yellow line indicates the baseline performance from the protected model on the train/test set.

to 318M. The overhead renders the attack impractical in real-world scenarios. Nevertheless, we pursued this experiment to evaluate whether FC layers could effectively learn the scaling introduced by Hadamard layers. This process entailed Fine-Tuning a ResNet-18 [24] model on the test set using the identical learning rate used in model training. As illustrated in Fig. 8, the substitution of protection layers with FC layers attains peak performance on the test set depicted by the green line, surpassing the baseline test accuracy depicted in yellow. However, the attack is unable to achieve performances close to the baseline accuracy of *80.48%* (depicted as the red line) for the training set, represented by the blue line. Even after 25 epochs, the accuracies of the training set persist at low levels of approximately *35%*, corresponding to a decline of roughly *45%*. This indicates that the model is over-fitting to the test set. We conclude that replacing the protection layers with FC layers is not a viable option to circumvent the protection.

E. DNNShield 's Runtime

To measure the runtime overhead of DNNShield , we trained three models for 10 epochs while averaging their time per epoch. We first trained the CNN model on the CIFAR-10 [29] dataset without protection layers. The mean time (seconds) per epoch was *3.1384s* with a variance of *0.031*. The model protected using two Hadamard layers had a mean time per epoch of *3.2745s* with a variance of *0.019*. The model protected using two Permutation layers had a mean time per epoch of *3.5352s* with a variance of *0.052*. These results demonstrate that the overhead introduced by DNNShield is minimal, e.g., less than *5%* for the model protected using Hadamard layers.

VII. DISCUSSION

The following section presents a discussion on the placement, combination, and amount of protection layers. Additionally, we will discuss the publication of the key.

Protection Layer Placement. The placement of protection layers within a DNN can impact its robustness against adversarial attacks. Placing them after activation functions can make it more difficult for adversaries to merge them into the preceding unprotected layer. This is because activation functions introduce non-linearity, making it more challenging to integrate them into protection layers.

Protection Layer Combination. The Hadamard and Permutation layers (cf. Sect. IV-B) are two distinct instantiations that offer unique advantages. The Hadamard layer introduces

a scaling pattern. The permutation layer, on the other hand, changes the order of output values. Combining both types of protection layers within a single DNN might further complicate the task of removing the protection layers for an adversary.

Protection Layer Amount. The number of protection layers used in a DNN has a direct resilience impact. Adding more protection layers can improve the robustness of the protection methods and increase the confidence in the model identifier. However, too many protection layers can lead to drawbacks, such as longer runtime. Therefore, achieving the desired level of robustness and identification confidence requires a balanced number of protection layers.

Key Publication. To facilitate efficient ownership verification, the model owner must establish that the key integrated into the model was indeed chosen by them and not another party. Therefore, the key or the hash of it should be publicly disclosed prior to model training, such as on a platform that assigns verifiable timestamps and links the key to the owner of the model, e.g. a blockchain [18].

VIII. Related Work

This section presents DNN IP protection works related to DNNShield .

Watermarking. Watermarking (WM) (cf. Sect. II) is a related approach to IP protection for DNNs. White-box WM schemes [47], [31], [43], [50], [51], [14], which directly modify and analyze the model weights, are among the most prevalent DNN WM techniques. These approaches typically necessitate a secret key and a secret watermark that can be extracted using this key. Consequently, the approaches can only be employed once, as the secret key must be revealed for ownership verification. Furthermore, watermark embedding is primarily accomplished through the use of an additional loss function, which increases training complexity. In contrast, our approach does not rely on any secrets, enabling limitless ownership verification. Additionally, the approach does not significantly increase training overhead as it does not introduce additional trainable parameters.

The second category, Black-box WM schemes [34], [2], [54], [32], [21], [27], involve feeding specific input samples into the model and analyzing their outputs. These inputs, function as the secret keys and are kept confidential. As a result, often the same limitations apply to these approaches. To incorporate the watermark, the training data must be manipulated which could degrade the models performance.

Fingerprinting. Fingerprinting (FP) extracts unique identifiers from trained DNNs for ownership verification. The methods can be categorized as parameter-based or input-based. Both have similar limitations than WM schemes. Parameter-based FP [14], [9] relies on a secret key, limiting its applicability. In contrast, DNNShield does not rely on secrecy, enabling an arbitrary number of verifications.

Input-based FP [53], [35], [6] utilizes distinct output patterns generated by different DNNs for specific inputs to identify

a model. However, this approach requires generating carefully crafted inputs, which can be used for adaptive attacks and, thus, need to be kept secret. Additionally, the identifier is only extracted after model training, leaving intermediate model states, known as checkpoints, susceptible to misuse by adversaries. In contrast, DNNShield provides a static identifier, unchanged during model training, addressing these limitations. DeepJudge [10], a testing framework proposed as alternative to traditional WM techniques, faces similar limitations. It compares the behavioral similarities between a trained DNN and a potentially infringing model based on six metrics. These metrics are derived from selected inference samples that capture the models' characteristics. However, the output for these specific samples could be manipulated and, contrary to DNNShield , the method cannot safeguard intermediate model states, leaving them susceptible to misuse.

Passporting. Fan *et al.* [20], [19] introduces passport layers as a DNN model protection mechanism. However, this approach is limited to specific convolutional layer configurations, namely pairs of convolutional layers followed by a normalization layer. The normalization layers are modified to form passport layers, functioning similar to our Hadamard layers. These layers incorporate trainable parameters that are optimized alongside the remaining model parameters and constrained by an additional loss. Therefore, the training complexity is increased and the application of the method results in training overhead. Furthermore, it prevents the protection of intermediate model checkpoints, as the passport layer parameters change throughout training. In contrast, DNNShield can be applied to both convolutional and Fully-Connected layers, regardless of subsequent layers, without introducing additional trainable parameters. Moreover, it safeguards intermediate checkpoints. Zhang *et al.* [56] introduce a method that incorporates a secret passport layer during training alongside the unprotected model. After releasing the unprotected model, the secret passport layers can be employed analogously to a secret key to verify ownership. This is because the unmodified model, when equipped with the secret passport layer, exhibits unique behavior. A similar approach is also proposed as an alternative in DeepIPR [20]. In contrast, our approach does not rely on any secret keys. Additionally, Chen *et al.* [11] demonstrated that DeepIPR's passport layers are vulnerable to ambiguity attacks. These attacks attempt to falsely claim ownership by replacing the passport's trainable parameters with different ones. A small portion (10%) of training data is leveraged with additional fully connected layers to identify alternative parameters. We have shown that finding alternative keys that have significant dissimilarity is infeasible for our approach. Furthermore, an adversary could attempt to circumvent the protection mechanism by replacing the protection layers with fully connected layers. However, our experiments in Sect. VI-D have shown that this strategy is ineffective against DNNShield and does not scale well in terms of model size.

Hardware-based IP protection. Hardware-based IP protec-

tion methods can be broadly classified into two categories: Hardware-Level IP protection [8] employs trusted execution environments (TEEs) to ensure that only approved models can be executed on specific hardware devices. A TEE acts as a secure enclave within the hardware, safeguarding sensitive information. Only models that match the secret model identifier are granted access to the hardware. Additionally, Hardware-Assisted IP protection [1] involves encrypting portions of the model and executing them within a TEE. During inference in the TEE a secret key is employed. In contrast, DNNShield eliminates the need for dedicated hardware or secret keys.

IX. Conclusion

Protecting the IP rights of DNN creators is a significant challenge in the rapidly evolving field of ML. Current approaches modify the training dataset, rely on the secrecy of an embedded key, introduce additional training parameters, restrict themselves to specific layer types, or leave unfinished model checkpoints unprotected. To address these limitations, we introduce DNNShield, a novel protection method that integrates protection layers into the model's architecture. These layers allow for accurate identification of the model, enabling ownership claims by the creator. Our approach includes two instances of protection layers, both demonstrating high resilience against fine-tuning, pruning, and sophisticated adaptive adversarial attacks while incurring negligible performance and runtime overhead. We extensively evaluated the approach across three datasets and three model architectures, confirming the efficacy of DNNShield in safeguarding DNNs.

REFERENCES

[1] C. Abhishek, M. Ankit, and S. Ankur, "Hardware-Assisted Intellectual Property Protection of Deep Learning Models," *DAC*, 2020.

[2] Y. Adi, C. Baum, M. Cisse, B. Pinkas, and J. Keshet, "Turning Your Weakness into a Strength: Watermarking Deep Neural Networks by Backdooring," *USENIX Security*, 2018.

[3] E. Benevento, D. Aloini, and N. Squicciarini, "Towards a real-time prediction of waiting times in emergency departments: A comparative analysis of machine learning techniques," *IJF*, 2023.

[4] Y. Bengio, P. Frasconi, and P. Simard, "The problem of learning long-term dependencies in recurrent networks," *ICNN*, 1993.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *NeurIPS*, 2020.

[6] X. Cao, J. Jia, and N. Z. Gong, "IPGuard: Protecting Intellectual Property of Deep Neural Networks via Fingerprinting the Classification Boundary," *ASIACCS*, 2021.

[7] C.-Y. Chang and S.-J. Su, "A neural-network-based robust watermarking scheme," *SMC*, 2005.

[8] H. Chen, C. Fu, B. D. Rouhani, J. Zhao, and F. Koushanfar, "DeepAttest: An End-to-End Attestation Framework for Deep Neural Networks," *ISCA*, 2019.

[9] H. Chen, B. D. Rouhani, C. Fu, J. Zhao, and F. Koushanfar, "DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models," *ICMR*, 2019.

[10] J. Chen, J. Wang, T. Peng, Y. Sun, P. Cheng, S. Ji, X. Ma, B. Li, and D. Song, "Copy, Right? A Testing Framework for Copyright Protection of Deep Learning Models," *IEEE SP*, 2022.

[11] Y. Chen, J. Tian, X. Chen, and J. Zhou, "Effective Ambiguity Attack Against Passport-based DNN Intellectual Property Protection Schemes through Fully Connected Layer Substitution," *CVPR*, 2023.

[12] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural Language Processing (almost) from Scratch," *JMLR*, 2011.

[13] P. Contributors, "TORCH.MUL," 2023, https://pytorch.org/docs/stable/generated/torch.mul.html.

[14] B. Darvish Rouhani, H. Chen, and F. Koushanfar, "DeepSigns: An End-to-End Watermarking Framework for Ownership Protection of Deep Neural Networks," *ASPLOS*, 2019.

[15] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research," *IEEE Signal Processing Magazine*, 2012.

[16] A. S. Dhanjal and W. Singh, "A comprehensive survey on automatic speech recognition using neural networks," *Multimedia Tools and Applications*, 2023.

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv preprint arXiv:2010.11929*, 2021.

[18] G. Estevam, L. M. Palma, L. R. Silva, J. E. Martina, and M. Vigil, "Accurate and decentralized timestamping using smart contracts on the Ethereum blockchain," *Inf Process Manag.*, 2021.

[19] L. Fan, K. W. Ng, and C. S. Chan, "Rethinking Deep Neural Network Ownership Verification: Embedding Passports to Defeat Ambiguity Attacks," *NeurIPS*, 2019.

[20] L. Fan, K. W. Ng, C. S. Chan, and Q. Yang, "DeepIPR: Deep Neural Network Ownership Verification With Passports," *TPAMI*, 2022.

[21] J. Guo and M. Potkonjak, "Watermarking Deep Neural Networks for Embedded Systems," *ICCAD*, 2018.

[22] D. Han, Z. Wang, W. Chen, K. Wang, R. Yu, S. Wang, H. Zhang, Z. Wang, M. Jin, J. Yang, X. Shi, and X. Yin, "Anomaly Detection in the Open World: Normality Shift Detection, Explanation, and Adaptation," *NDSS*, 2023.

[23] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both Weights and Connections for Efficient Neural Networks," *NeurIPS*, 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CVPR*, 2016.

[25] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for MobileNetV3," *ICCVW*, 2019.

[26] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[27] H. Jia, C. A. Choquette-Choo, V. Chandrasekaran, and N. Papernot, "Entangled Watermarks as a Defense against Model Extraction," *USENIX Security*, 2021.

[28] T. Krauß and A. Dmitrienko, "MESAS: Poisoning Defense for Federated Learning Resilient against Adaptive Attackers," *CCS*, 2023.

[29] A. Krizhevsky, G. Hinton *et al.*, "Learning Multiple Layers of Features from Tiny Images," *Citeseer*, 2009.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NeurIPS*, 2012.

[31] Y. Li, B. Tondi, and M. Barni, "Spread-Transform Dither Modulation Watermarking of Deep Neural Network," *JISA*, 2021.

[32] Z. Li, C. Hu, Y. Zhang, and S. Guo, "How to Prove Your Model Belongs to You: A Blind-Watermark Based Framework to Protect Intellectual Property of DNN," *ACSAC*, 2019.

[33] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep Learning for Generic Object Detection: A Survey," *IJCV*, 2020.

[34] E. L. Merrer, P. Pérez, and G. Trédan, "Adversarial Frontier Stitching for Remote Neural Network Watermarking," *Neural Computing and Applications*, 2019.

[35] L. Nils, Z. Yuxuan, and K. Florian, "Deep Neural Network Fingerprinting by Conferrable Adversarial Examples," *ICLR*, 2021.

[36] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *NeurIPS*, 2019.

[38] P. Rieger, T. Krauß, M. Miettinen, A. Dmitrienko, and A.-R. Sadeghi, "CrowdGuard: Federated Backdoor Detection in Federated Learning," *arXiv preprint arXiv:2210.07714*, 2023.

[39] S. Shan, W. Ding, H. Zheng, and B. Y. Zhao, "Post-Breach Recovery: Protection against White-Box Adversarial Examples for Leaked DNN Models," *CCS*, 2022.

[40] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition." *ICLR*, 2015.

[41] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, 2012.

[42] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, "Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?" *IEEE TMI*, 2016.

[43] E. Tartaglione, M. Grangetto, D. Cavagnino, and M. Botta, "Delving in the loss landscape to embed robust watermarks into neural networks," *ICPR*, 2021.

[44] The Linux Foundation, "Pytorch," 2022, https://pytorch.org.

[45] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The Computational Limits of Deep Learning," *arXiv preprint arXiv:2007.05558*, 2022.

[46] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," *arXiv preprint arXiv:2302.13971*, 2023.

[47] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding Watermarks into Deep Neural Networks," *ICMR*, 2017.

[48] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arXiv preprint arXiv:1706.03762*, 2023.

[50] J. Wang, H. Wu, X. Zhang, and Y. Yao, "Watermarking in Deep Neural Networks via Error Back-propagation," *Electronic Imaging*, 2020.

[51] T. Wang and F. Kerschbaum, "RIGA: Covert and Robust White-Box Watermarking of Deep Neural Networks," *WWW*, 2021.

[52] C. Xie, P. Yi, B. Zhang, and F. Zou, "DeepMark: Embedding Watermarks into Deep Neural Network Using Pruning," *ICTAI*, 2021.

[53] K. Yang, R. Wang, and L. Wang, "MetaFinger: Fingerprinting the Deep Neural Networks with Meta-training," *IJCAI*, 2022.

16

[54] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting Intellectual Property of Deep Neural Networks with Watermarking," *ASIACCS*, 2018.

[55] J. Zhang, D. Chen, J. Liao, H. Fang, W. Zhang, W. Zhou, H. Cui, and N. Yu, "Model Watermarking for Image Processing Networks," *AAAI*, 2020.

[56] J. Zhang, D. Chen, J. Liao, W. Zhang, G. Hua, and N. Yu, "Passport-aware Normalization for Deep Model Protection," *NeurIPS*, 2020.

APPENDIX A
HADAMARD KEY MERGE INTO LINEAR LAYER

In case no activation function is utilized after calculation of the linear layers output it is straightforwardly possible to merge the key of the Hadamard layer into the weights of the linear layer. Assume an input vector x, a linear layer with weights w and bias b, followed by a Hadamard protection layer with key k. The forward pass is computed as $y = (x \cdot w^t + b) \times k$ which expands to $(\sum_i x_{i,j} \cdot w_{i,j} + b_j) \times k_j$, where i denotes the i-th and j denotes the j-th column in the matrix. Now an adversary can compute $w'_{i,j} = w_{i,j} \times k_j$ and $b'_j = b_j \times k_j$ to merge the key with the linear layer. The weights and biases of the new linear layer are now specified by w' and b'.

APPENDIX B
HADAMARD KEY EXTRACTION FROM MERGED LINEAR LAYER

The key can be extracted from a merged linear layer by calculating $\frac{w'_{i,j}}{w_{i,j}} = k_{i,j}$. For each key value multiple values are obtained, therefore, an averaging mechanism is required to obtain a single value. Also the key from the bias is extracted by calculating $\frac{b'_j}{b_j} = k_j$. To enhance the robustness and perform aggregation, outliers of three times the standard deviation from the mean are removed and then the mean of the remaining values is computed. The key from the weights and bias are equally weighted and averaged. Therefore, the resulting key can be compared as described in Sect. IV.

APPENDIX C
HADAMARD KEY MERGE INTO CONVOLUTIONAL LAYER

Consider a convolutional layer with one filter with weights f, a kernel size of 2 by 2 and for simplicity without bias, after calculation of the convolution layer a Hadamard layer with key k is applied. The forward pass calculates $y = (x * f)$, where $*$ denotes the cross-correlation operation and x is the input vector as shown in Fig. 9. The value from the convolutions output are calculated as $x_1 * f_1 + x_2 * f_2 + x_4 * f_3 + x_5 * f_4$, the second cell is calculated as $x_2 * f_1 + x_3 * f_2 + x_5 * f_3 + x_6 * f_4$. Now to merge the key layer into the first convolutional filters weight one would need to calculate $f'_1 = f_1 \times k_1$ and $f'_1 = f_1 \times k_2$. As $k_1 \neq k_2$ it is not possible to merge the key layer into the convolutional filters weights. This assumes that the values of the keys are not equivalent, which is highly probable given that the values are randomly generated.

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{bmatrix}$$

Fig. 9: Input Matrix x.

APPENDIX D
MODEL ARCHITECTURES

The approaches were assessed across different model architectures, encompassing Fully Connected and Convolutional models, alongside the widely used ResNet-18 [24] architecture.

Fully Connected The Fully Connected Network (FCN) encompasses linear layers and the ReLu activation function as shown in Table IV. The model can only be protected using Hadamard protection layers.

Layer	Input Width and Height	Input Channels
Linear 1	32x32	3
Protection Layer 1	1	15
ReLu	1	15
Linear 2	1	15
Protection Layer 2	1	10
ReLu	1	10
Linear 3	1	10

TABLE IV: Architecture of the Fully Connected Network. The dimensions are calculated based on usage with the CIFAR-10 [29] dataset.

Convolutional The architecture of the convolutional model is shown in Table V. The convolutional layers (Conv2d) have kernel sizes of 5 by 5 and the maxpooling layers (MaxPool2d) have kernels of size 2 by 2. In case the model is protected by Permutation layers only the Protection Layer 1 and Protection Layer 2 are utilized, as the Permutation layers can not be used in conjunction with Linear layers.

Layer	Input Width and Height	Input Channels
Conv2d	32x32	3
Protection Layer 1	28x28	32
ReLu	28x28	32
MaxPool2d	28x28	32
Conv2d	14x14	32
Protection Layer 2	10x10	64
ReLu	10x10	64
MaxPool2d	10x10	64
Linear	5x5	64
Protection Layer 3	1	512
ReLu	1	512
Linear	1	512
Protection Layer 4	1	256
ReLu	1	256
Linear	1	256

TABLE V: Architecture of the Convolutional Neural Network (CNN). The dimensions are calculated based on usage with the CIFAR-10 [29] dataset.

ResNet-18 The architecture of the protected ResNet-18 [24] model is shown in Table VI. The model consists of convolutional layers (Conv2d), batch normalization layers (Batch-Norm), and skip connections (SkipConnection) which add the input of previous layers to following layers. We protect the ResNet-18 [24] model with nine well-distributed Protection layers as shown in Table VI. For Permutation Layer Protection only the first seven Protection Layers are used because protection layers 8 and 9 consists of a single value per channel

which can not be permuted. The Hadamard protection layers' key values represent *0.28%* of the total parameters, while the Permutation protection layers' key values represent only *0.008%* of the total parameters.

Layer	Input Width and Height	Input Channels
Conv2d	32x32	3
Protection Layer 1	16x16	64
BatchNorm	16x16	64
ReLu	16x16	64
MaxPool2d	16x16	64
Conv2d	8x8	64
Protection Layer 2	8x8	64
BatchNorm	8x8	64
ReLu	8x8	64
Conv2d	8x8	64
BatchNorm	8x8	64
ReLu	8x8	64
Conv2d	8x8	64
Protection Layer 3	8x8	64
BatchNorm	8x8	64
ReLu	8x8	64
Conv2d	8x8	64
BatchNorm	8x8	64
ReLu	8x8	64
Conv2d	8x8	64
Protection Layer 4	4x4	128
BatchNorm	4x4	128
ReLu	4x4	128
Conv2d	4x4	128
BatchNorm	4x4	128
SkipConnection	4x4	128
ReLu	4x4	128
Conv2d	4x4	128
Protection Layer 5	4x4	128
BatchNorm	4x4	128
ReLu	4x4	128
Conv2d	4x4	128
BatchNorm	4x4	128
ReLu	4x4	128
Conv2d	4x4	128
Protection Layer 6	2x2	128
BatchNorm	2x2	256
ReLu	2x2	256
Conv2d	2x2	256
BatchNorm	2x2	256
SkipConnection	2x2	256
ReLu	2x2	256
Conv2d	2x2	256
Protection Layer 7	2x2	256
BatchNorm	2x2	256
ReLu	2x2	256
Conv2d	2x2	256
BatchNorm	2x2	256
ReLu	2x2	256
Conv2d	2x2	256
Protection Layer 8	1x1	256
BatchNorm	1x1	512
ReLu	1x1	512
Conv2d	1x1	512
BatchNorm	1x1	512
SkipConnection	1x1	512
ReLu	1x1	512
Conv2d	1x1	512
Protection Layer 9	1x1	512
BatchNorm	1x1	512
ReLu	1x1	512
Conv2d	1x1	512
BatchNorm	1x1	512
ReLu	1x1	512
AdaptiveAvgPool2d	1x1	512
Linear	1	512

TABLE VI: Architecture of the ResNet-18 [24] model with Protection layers after first Convolution. The dimensions are calculated based on the CIFAR-10 [29] dataset.

APPENDIX E
ADDITIONAL FUNCTIONALITY EXPERIMENTS

To showcase the robustness of the protected ResNet-18 [24], it was protected using Hadamard Protection layers and Permutation Protection layers. The results are depicted in Fig. 10, the first row shows the results for the model protected by Hadamard layers, while the second row shows the results for the model protected by Permutation layers. For each model type three experiments identical to the ones described in Sect. VI-A were conducted. The first evaluation (depicted in the first column of Fig. 10) assigned randomly generated false
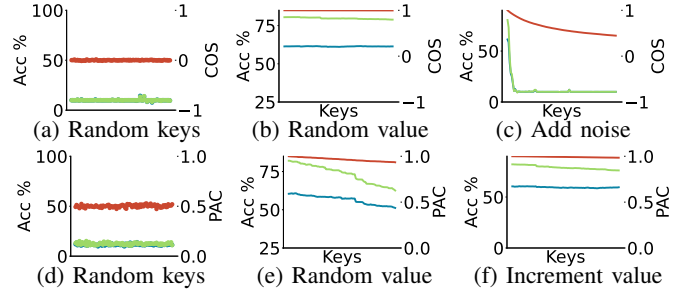

Fig. 10: ResNet-18 [24] robustness protected by Hadamard/Permutation layers to key manipulation, e.g. random key, replacing of key value, and, adding noise to key. Red line depicts key similarity, Green and Blue depict train and test accuracy.
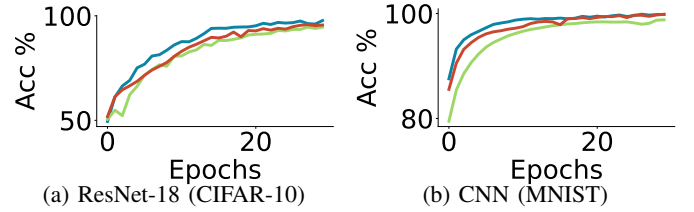

Fig. 11: The training set accuracy of three different models at each training epoch are visualized. The blue line represents the unprotected model, the red line the model protected using Permutation Layers and the green line the model protected using Hadamard layers.

keys to the protected models and measured the performance and key similarity. The second experiment (second column in Fig. 10) iteratively replaced from zero up to 100 values, within the key, with random values. The third experiment (depicted in the third column of Fig. 10 added noise to the key or incremented the key by one for the model protected using Permutation layers. All results indicate a strong robustness to key manipulation attempts similar to the CNN model.

APPENDIX F
ADDITIONAL FIDELITY EXPERIMENTS

In the following we present additional fidelity experiments for the ResNet-18 [24] and CNN model trained on CIFAR-10 [29] and MNIST [15]. The results, visualized in Fig. 11, depict the accuracies for the training set for the unprotected model at each training epoch as the blue line, the green line depict the model protected using Hadamard layers, and the red line the model protected using Permutation layers.